Algorithmes de la morphologie mathématique pour les architectures orientées flux

Jaromír Brambor

MINES PARIS
Collège doctoral

N° attribué par la bibliothèque

|_|_|_|_|_|_|_|_|_|_|_|

THÈSE

pour obtenir le grade de
Docteur de l'École des Mines de Paris
Spécialité "Morphologie Mathématique"

présentée et soutenue publiquement
par
Jaromír BRAMBOR

le 11 Juillet 2006

ALGORITHMES DE LA MORPHOLOGIE MATHÉMATIQUE POUR LES ARCHITECTURES ORIENTÉES FLUX

Directeur de thèse : Michel BILODEAU

Jury

M.	*Gilles*	*BERTRAND*	*Examinateur*
M.	*Michel*	*BILODEAU*	*Examinateur*
M.	*Pierre*	*BONTON*	*Rapporteur*
M.	*Laurent*	*CHABIN*	*Examinateur*
M.	*Dominique*	*JEULIN*	*Président*
M.	*Michel*	*PAINDAVOINE*	*Rapporteur*

Marques commerciales déposées et/ou utilisées dans un ou plusieurs pays et leurs propriétaires respectifs

Marques	Propriétaire
Aphelion	Amerinex Applied Imaging, Inc. et ADCIS SA.
AMD, AMD Athlon, AMD Opteron	Advanced Micro Devices, Inc.
ARM, ARM11	ARM Limited
ATI, CrossFire, Radeon	ATI Technologies Inc.
SuperH	Hitachi Ltd.
IBM, PowerPC	International Business Machines Corporation.
IEEE, POSIX	The Institute of Electrical and Electronics Engineers, Incorporated.
Intel, Itanium, MMX, Pentium	Intel Corporation
MATLAB	The MathWorks, Inc.
DirectX, Microsoft, Microsoft Press, Windows, Win32, Xbox	Microsoft Corporation
AltiVec, MOTOROLA,	Motorola, Inc.
GeForce, NVidia, NVIDIA Quadro, SLI	NVIDIA Corporation
PCI Express	PCI-SIG
RenderMan	Pixar Animation Studios
OpenGL, OpenGL ES	Silicon Graphics Incorporated
Sony, PlayStation	Sony Corporation
Sun, VIS	Sun Microsystems, Inc.
Crusoe, Efficeon, Transmeta	Transmeta Corporation
TSMC	Taiwan Semiconductor Manufacturing Company, Ltd.
Linux	Linus Torvalds
Wikipedia	Wikimedia Foundation, Inc.

À
Jaromír et Blanka
mes parents

et à
Julie et Maxim Jaromír
ma famille

Cette page est blanche par intention

Remerciements

Cette thèse est l'aboutissement d'un projet de recherche de plusieurs années. Ce projet doit un grand merci à tous ceux qui l'ont soutenu directement ou indirectement lors de sa réalisation et à tous ceux qui ont suivi la création et la finalisation de cette thèse, de loin ou de près.

Au Centre de Morphologie Mathématique et à ses membres

J'aimerais exprimer mes sincères remerciements à l'École Nationale Supérieure des Mines de Paris et en particulier à son Centre de Morphologie Mathématique à Fontainebleau représenté par Monsieur Fernand Meyer pour m'avoir accueilli et m'avoir permis de travailler sur le sujet de mon projet de recherche.

Mes remerciements s'adressent également à Monsieur Michel Bilodeau, mon directeur de thèse, pour le temps qu'il a consacré à l'encadrement de mon projet et de ma thèse et pour ses avis sur mon travail.

Un merci s'adresse à Monsieur Jean Serra que j'apprécie beaucoup pour son travail effectué dans le domaine de la morphologie mathématique. Pour son travail qui influence le traitement d'images depuis des décennies, qui conduit d'autres chercheurs et qui les incite à utiliser la morphologie mathématique et à continuer de la développer.

Un merci aussi aux autres membres du CMM – Beatriz Marcotegui et Dominique Jeulin, Serge Beucher, Jean-Claude Klein, Petr Dokládal, Etienne Decencière, Francis Bach et John-Richard Ordóñez Varela pour leur présence et leurs points de vue discutés au cours de séances de lectures et d'exposés car dans la science et la recherche, c'est aussi l'effet synergique qui importe et qui permet d'avancer plus rapidement.

Je ne dois pas oublier de remercier deux membres non scientifiques mais dont l'importance au sein du CMM est cruciale et chez qui j'apprécie leurs valeurs humaines et leur serviabilité. Mon merci s'adresse ainsi à Catherine Moysan et à Laura Andriamasinoro.

Aux partenaires du projet de recherche

Une partie de cette thèse a été développée dans le cadre du projet *Medea+ PocketMM* et financée par le Ministère de l'Économie, des Finances et de l'Industrie (MINÉFI) de la République française. Un grand merci à nos partenaires et en particulier à STMicroelectronics pour nous avoir procuré les outils de développement.

À ma famille

Un merci spécialement chaleureux à ma famille représentée par ma femme Julie et mon fils Maxim Jaromír, par Blanka, ma mère, in memoriam par Jaromír, mon père, par mes sœurs Milena, Blanka et Soňa et leurs familles pour la patience dont ils ont fait preuve lors de la rédaction du manuscrit et pour le soutien qu'ils m'ont prodigué dans les moments rudes et éprouvants. Merci beaucoup.

À mes collègues en recherche

Mes collègues du Centre de Morphologie Mathématique méritent, eux aussi, une part de remerciement car c'est eux qui ont donné l'esprit multiculturel à mon séjour à Fontainebleau et qui m'ont aidé à mettre au point mon français oral à travers des discussions portant sur les sujets scientifiques et métaphysiques. Mes mercis s'adressent[1] à Eva Dokládalová, Mathilde Boehm, à Thomas Walter, Thomas Retornaz, Raffi Enficiaud, Costin Alin Caciu, Jesús Angulo, Romain Lerallut, Gabriel Fricout, Timothée Faucon, Thibauld Nion, Nicolas Laveau, Souhaïl Outal et Maxime Moreaud.

[1] nommés par ordre croissant selon la distance topologique à partir de mon bureau

ALGORITHMES DE LA MORPHOLOGIE MATHÉMATIQUE
POUR LES ARCHITECTURES ORIENTÉES FLUX

École des Mines de Paris

Jaromír BRAMBOR, 11 Juillet 2006

Résumé

Mots clés : morphologie mathématique, algorithmes rapides, flux de données, macro blocs SIMD, processeurs graphiques, Haskell, description formelle, lambda calcul

Cette thèse est consacrée aux algorithmes de morphologie mathématique qui peuvent considérer les pixels d'une image comme un flux de données. Nous allons démontrer qu'un grand nombre d'algorithmes de morphologie mathématique peuvent être décrits comme un flux de données traversant des unités d'exécution. Nous verrons que cette approche peut aussi fonctionner sur des processeurs génériques possédant un jeu d'instructions multimédia ou sur des cartes graphiques.

Nous présentons dans un premier temps les possibilités offertes par les architectures grand public en guise de traitements morphologiques et explorons ensuite leurs capacités d'exécution parallèle avec des jeux d'instructions SIMD. Nous terminons cette partie en examinant les capacités de calcul parallèle à l'échelle des tâches et des fils d'exécution que nous apportent les processeurs à plusieurs cœurs.

Pour décrire les algorithmes en flux de données, nous proposons d'utiliser le langage fonctionnel Haskell, ce qui nous permettra de décrire les briques de base de la construction des algorithmes de morphologie mathématique. On applique ces briques dans la description des algorithmes les plus couramment utilisés (dilatation/érosion, opérations géodésiques, fonction distance et nivellements) ce qui facilitera le portage de ces algorithmes sur plusieurs plate-formes.

L'apport principal de cette thèse est qu'elle aborde un sujet important du point de vue pratique et qu'elle explore les capacités SIMD des architectures multimédia en vue d'assurer l'exécution rapide des algorithmes morphologiques travaillant sur le voisinage. Nous proposons pour la construction des algorithmes morphologiques un mode d'exécution original par macro blocs et nous étudions en profondeur la transposition de cette idée aux architectures SIMD. Nous montrons que l'utilisation des macro blocs est particulièrement intéressante pour les architectures multimédia et nous montrons également que les algorithmes morphologiques proposés dans cette thèse atteignent de meilleures performances que les implémentations standard. Un nouveau champ s'ouvre ainsi aux algorithmes développés dans les applications de traitement d'images en temps réel.

Cette thèse explore également les processeurs graphiques et démontre sur des résultats expérimentaux qu'ils sont, dès à présent, assez performants pour concurrencer les processeurs généraux. L'apport secondaire de cette thèse est celui du formalisme fonctionnel basé sur le lambda calcul qui a été adopté pour la description des algorithmes travaillant sur les flux de données.

Cette page est blanche par intention

ALGORITHMS OF MATHEMATICAL MORPHOLOGY
FOR STREAM-ORIENTED ARCHITECTURES

École des Mines de Paris
Jaromír BRAMBOR, 11 July 2006

Abstract

Keywords : Mathematical Morphology, fast algorithms, stream of data, SIMD macro blocs, graphics processors, Haskell, formal description, lambda calculus

This thesis deals with the algorithms of Mathematical Morphology that can consider the pixels of an image as if they were a stream of data. We will show that a great number of algorithms of Mathematical Morphology can be described by data flows (streams) passing through the operating units. We will see that this approach can function on generic processors supporting a multi-media instruction set as well as on graphics cards.

Firstly, we present the possibilities of main stream architectures for morphological processing and then we explore their capacities in terms of parallel execution with SIMD instruction sets. Finally, we examine the possibilities of parallel computing using tasks and threads on multi-core processors.

We propose to use the functional language Haskell for the description of algorithms operating on data flows. This allows us to describe the building blocks that are used to construct morphological algorithms. We apply these bulding blocks in the description of the most usually used algorithms (dilation/erosion, geodesic operations, distance function and levelings). This will also facilitate the porting of these algorithms onto several platforms.

The principal contribution of this thesis is to address an important subject from the practical point of view and to explore SIMD capabilities of multi-media architectures to ensure the fast execution of morphological algorithms working on neighborhood. We propose an original mode of execution by macro blocks for the construction of morphological algorithms and we study in depth the transposition of this idea to SIMD architectures. We show that the use of macro blocks is particularly interesting for multi-media architectures. We also show that the morphological algorithms proposed in this thesis reach better performances than standard implementations. Thus, a new field opens to algorithms we have developed here in real-time image processing applications.

This thesis also explores the graphics processors and shows on experimental results that they are already powerful enough to compete with general processors. The secondary contribution of this thesis is the functional formalism based on lambda calculus that is used to describe algorithms working on data flows.

Cette page est blanche par intention

Table des matières

Guide de thèse

Pour accentuer la bonne lisibilité de cette thèse et pour permettre au lecteur de se créer une vision globale du contenu avant même d'entamer la lecture, nous voudrions brièvement exposer à cette place les points clés de la thèse accompagnés de quelques remarques. Ces quelques lignes vont, comme nous le croyons, servir à une meilleure orientation dans l'ensemble de ce traité.

Cette thèse étudie la problématique du calcul de la morphologie mathématique sur les architectures destinées à opérer sur les flux des données (streams) dont nous explorons deux groupes différents. Le premier groupe englobe les architectures pour le calcul général avec les fonctionnalités SIMD, le deuxième les architectures des processeurs graphiques qui travaillent naturellement avec les flux de données et qui ont une structure très particulière. Nous nous intéressons aux algorithmes rapides et utilisables en pratique sur les deux groupes d'architectures et nous décrivons plusieurs algorithmes originaux que nous avons pu développer.

Le document est divisé en deux parties principales derrière lesquelles nous ajoutons une conclusion générale suivie par les annexes. Nous commençons notre exposé par la partie nommée *Introduction, bases théoriques et appareil mathématique* où nous introduisons tout d'abord la problématique des architectures, avec un accent sur le pipeline graphique et les processeurs graphiques. Ensuite, nous présentons le formalisme mathématique fonctionnel. Tout le reste de cette thèse s'appuie fortement sur ce formalisme. Il trouvera son utilité dans la description des algorithmes et nous verrons que son grand avantage se situe dans sa façon de modéliser qui permet de décrire d'une façon abstraite même les algorithmes étroitement liés aux fonctionnements d'une architecture particulière. Dans cette même partie, nous introduirons également les outils algorithmiques de base, exprimés à l'aide de formalisme fonctionnel. Il s'agit surtout des primitives pour l'organisation de données et pour le parcours d'une image. Nous ajouterons également les outils qui se focalisent directement à la morphologie mathématique et nous présenterons la manière formelle de leur expression. Ces outils de base sont utilisés dans la partie suivante, incorporés dans la description des algorithmes fonctionnels complexes.

La deuxième partie est consacrée à la description des algorithmes et les concepts algorithmiques. Elle est organisée par chapitres dédiés chacun à un thème particulier. Dans chaque chapitre, nous explorons les possibilités d'utilisation des architectures SIMD et des GPU pour les algorithmes relatifs à la morphologie mathématique.

Nous y présenterons tout d'abord les algorithmes morphologiques génériques et SIMD non-triviaux dont le traitement ne dépend pas d'un sens du parcours particulier, suivis par les algorithmes itératifs et qui travaillent en utilisant un parcours spécifique de l'image pour finir par la présentations des algorithmes pour les éléments structurants de la forme d'un segment qui vont combiner toutes les techniques présentés.

Nous abordons également des sujets que nous jugeons prometteurs pour l'avenir car ils se consacrent aux traitements morphologiques sur les processeurs graphiques. Le travail avec ces processeurs est particulier mais il a beaucoup en commun avec les traitements SIMD qui seront décrits préalablement. Dans les algorithmes pour les GPU nous utiliserons avec avantage le style de travail que nous décrivons pour les architectures SIMD et nous proposerons les algorithmes et les solutions pratiques adaptées au traitement sur les processeurs graphiques.

Nous décrivons également les notions de base pour pouvoir exprimer le coût d'un algorithme. Sachant que dans la pratique c'est surtout le coût exprimé en termes de temps du calcul qui nous intéresse le plus, nous présentons les techniques envisageables pour une telle étude de complexité.

Nous allons terminer cette thèse avec une *conclusion* générale présentant le sommaire des résultats obtenus et avec des *perspectives* qui ouvrent la possibilité d'une prochaine continuation des recherches portant sur le sujet traité.

L'annexe contient les définitions auxiliaires des fonctions en langage fonctionnel. Nous n'avons pas inclus ces définitions dans le texte principal pour deux raisons – nous avons estimé que leurs noms saisissaient bien le sens intuitif de leur comportement et on a voulu ne pas encombrer le texte principal avec des détails qui pourraient détourner l'attention du lecteur du sujet exposé. En même temps, pour présenter cette thèse en tant que conception complète, nous ne voulions pas les omettre et nous les incluons ainsi dans l'annexe.

Nous incitons le lecteur à utiliser la version électronique de cette thèse. Elle lui offre les possibilités hypertexte et facilite ainsi la liaison à partir du texte aux ressources bibliographiques, aux définitions des termes et surtout aux définitions des fonctions dans le formalisme fonctionnel. Surtout pour ce dernier, la version électronique peut s'avérer un outil très pratique qui peut, en cas de doute, considérablement améliorer l'orientation dans les algorithmes.

La bibliographie elle-même est également dotée des fonctions hypertexte qui permettent de visualiser directement les sources souhaitées, bien entendu les sources bibliographiques ayant une version en ligne que nous avons pu consulter.

Partie I

Introduction, bases théoriques et appareil mathématique

Cette page est blanche par intention

CHAPITRE 1

Motivation

1.1 Évolution en chiffres

Une étude des activités du secteur des circuits intégrés effectuée en 1965 par Gordon Moore[1] lui a permis d'estimer l'évolution de ce secteur comme une tendance exponentielle. Moore se basait sur le nombre de composantes par circuit intégré et il a inclus dans son observation les circuits mis sur le marché pendant les années 1959-1965.

Il a publié ensuite un article[Moo65] où il expliquait que les conditions pour l'évolution du secteur étaient favorables et a prévu, à l'horizon des 10 années suivantes, une croissance exponentielle par un facteur 2 chaque année.

La formule présentée par Moore exprimait une approximation des données mesurées et essayait de les extrapoler dans le temps. Les années suivantes ont vérifié que Moore avait eu raison en ce qui concerne la croissance exponentielle du nombre de composantes par circuit mais ont également corrigé le facteur 2 prévu par Moore pour chaque année à un facteur 2 pour tous les 18 à 24 mois. Cette formule est devenue célèbre, elle s'est inscrite dans l'histoire de l'informatique d'une façon informelle comme la Loi de Moore et elle avait, comme elle l'a toujours, un grand succès non seulement parmi le public populaire mais également dans le cercle scientifique, citons comme preuve la réédition de l'article original de Moore en 1998[Moo98] par l'organisation IEEE. Pour aller plus loin dans cette philosophie, divers auteurs déclinent cette loi sur d'autres types de données connexes aux processeurs. Il est courant d'utiliser les diagrammes de l'évolution de divers paramètres dépendant du temps. On parle ainsi de vecteurs de Moore[Gro02] et on les utilise pour démontrer les aspects plus détaillés de la production des processeurs tels que la lithographie, la consommation d'énergie, les performances, les chiffres d'affaires.

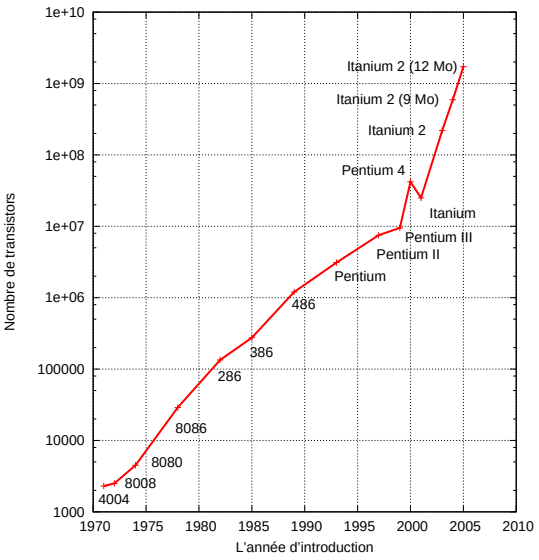


FIG. 1.1 : Évolution du nombre des transistors par produit Intel

Si nous regardons de près fig. 1.1[2], nous pourrons observer, depuis l'année 2000, un changement

[1] Gordon Moore devint en 1968 un des cofondateurs de l'Intel Corporation

[2] Données collectées à partir des publications[NSG05a, NSG05b, Boh04] et des documents de communication d'Intel

non négligeable de la tendance de la courbe qui correspondrait à présent (année 2005) à un facteur de croissance égal à 2 pour chaque année. Notre argumentation s'appuie ici sur un échantillon de 34 années de production d'un grand représentant du marché des processeurs grand public, un représentant qui est aussi très actif et très avancé[Boh04] dans l'innovation des technologies pour la fabrication des puces en silicium.

Ce nouveau dynamisme des possibilités de fabrication pourrait nous inciter à réfléchir à la direction que prendront les architectures grand public dans les années à venir. Là où l'utilisation d'une technologie de gravure de plus en plus précise[Boh04, Int06a] – 130 nm en production en 2001, 90 nm en production en 2003, 65 nm en production en 2005, 45 nm planifiée pour la production en 2007, 32 nm planifié en 2009[GS03] – est en train d'atteindre les dimensions de quelques couches d'atomes, on commence à parler de nanotechnologies et à rechercher des solutions pour la gravure à 13 nm[Boh02] et là où il est possible d'implémenter[NSG05a] 1 milliard et 720 millions de transistors sur une seule puce, cf. tab. 1.1, la parallélisation à l'intérieur des puces prend sa relève. Si on donne l'exemple des processeurs grand public, nous pourrons mentionner les technologies déjà présentes comme les architectures superscalaires ou les unités de traitement vectoriel, comme les architectures multithread avec un seul cœur ou les architectures multithread avec plusieurs cœurs. La dernière qui est également la plus jeune – la parallélisation par la multiplication du nombre des cœurs à l'intérieur d'un processeur sur une seule puce – est présentée par Intel comme son produit pilote, nommée Platform 2015[Int05], est adoptée en tant que stratégie d'évolution pour les dix prochaines années.

Année	Processeur	Nombre de transistors
1971	4004	2 300
1972	8008	2 500
1974	8080	4 500
1978	8086	29 000
1982	286	134 000
1985	386	275 000
1989	486	1 200 000
1993	Pentium	3 100 000
1997	Pentium II	7 500 000
1999	Pentium III	9 500 000
2000	Pentium 4	42 000 000
2001	Itanium	25 000 000
2003	Itanium 2	220 000 000
2004	Itanium 2 (9 Mo L3 cache)	592 000 000
2005	Itanium 2 (12 Mo L3 cache)	1 720 000 000

TAB. 1.1 : Évolution de nombre des transistors par produit Intel

1.2 Processeurs à usage général – les GPP et les GPPMM

Sous le terme de *processeurs à usage général* nous comprenons les processeurs les plus universels possible qui sont capables d'exécuter plusieurs types de calcul différents. L'universalité de ces processeurs est gagnée au préjudice de la spécialisation et par conséquent de la performance, même si celle-ci doit rester raisonnable vis-à-vis de l'utilité applicative de ces processeurs. Pour se référer à un tel processeur par la suite, nous allons utiliser l'abréviation *GPP*, dérivée d'un terme anglais *General Purpose Processor*.

Nous nous intéresserons aux processeurs de ce type, plus précisément à une catégorie de ces processeurs destinée principalement à l'usage des particuliers. C'est une catégorie très puissante en ce qui concerne le nombre d'unités vendues mais aussi une catégorie présentant certaines particularités. Notamment il s'agit de contraintes sur

- le prix qui doit être, pour des raisons économiques, plutôt en bas d'échelle,
- les dimensions qui ne doivent pas être excessives ou qui doivent être même les plus petites possible dans le cas où nous visons les produits portables,
- la consommation d'énergie qui est une contrainte imposée pour des raisons soit pratiques (besoin d'un système de refroidissement particulier ou non ?), soit financières (coût d'électricité) soit de durée d'autonomie de l'alimentation (produits portables).

Nous trouvons ces processeurs commercialisés sur le marché de grand public au cœur des ordinateurs professionnels, personnels ou portables, incorporés dans les produits nomadiques. Cependant, nous pouvons les trouver aussi bien éloignés du grand public dans les solutions non destinées à l'usage personnel tels que les serveurs informatiques ou également les processeurs embarqués et dans des applications variées, industrielles, civiles ou militaires.

Ce type de processeurs essaie de couvrir le plus grand nombre de clients potentiels et pour s'y adapter, leurs architectures ont bien évolué au cours des années. Elles nous proposent, dans leurs versions actuelles, les fonctionnalités dites multimédia[1] pour le traitement de données, ce qui veut dire qu'une partie de l'architecture est dédiée au traitement de données régulières d'un grand volume[2], telles que le son, les images, la vidéo. Nous parlons ainsi des processeurs à usage général avec les extensions multimédia, *GPPMM* (une abréviation dérivée d'un terme anglais *General Purpose Processor with MultiMedia extensions*). Le tableau 1.2 présente une liste non exhaustive des appellations des architectures ou des extensions multimédia selon divers fabricants de processeurs.

Fabricant	Fonctionnalités ou extensions multimédia
Intel IA-32	MMX, SSE, SSE2, SSE3
Intel IA-64	inclus
Intel PCA (PXA27x)	Wireless MMX
AMD	3DNow!, 3DNow!2
AMD64	inclus
Sun	VIS
MOTOROLA	AltiVec
SuperH	SHmedia
Transmeta Efficeon (VLIW)	MMX, SSE, SSE2, SSE3

TAB. 1.2 : Architectures multimédia

Ce sont les capacités multimédia qui vont nous intéresser dans cette thèse chez les GPPMM car en les utilisant, nous pouvons bénéficier très facilement et sur la plupart des architectures existantes grand public d'une réduction prévisible du temps de calcul et par dualité d'une augmentation du débit de traitement de données. De plus, les instructions multimédia pour le traitement par la morphologie mathématique sont semblables et nous pouvons ainsi obtenir une portabilité plus ou moins aisément. Cela est valable même pour des processeurs très différents, comme c'est le cas pour les processeurs puissants[LFB01] d'un côté et pour les processeurs à basse consommation[Bra02] de l'autre.

1.3 Processeurs graphiques – les GPU

Une place très particulière parmi les architectures des processeurs est détenue par les processeurs graphiques, les *GPU* (dérivé d'un terme anglais *Graphic Processing Unit*). Il s'agit des processeurs qui étaient initialement utilisés à l'accélération de la visualisation des scènes 3D, très exigeante en ce qui concerne le nombre d'opérations effectuées et la spécificité du calcul.

Le calcul sur les GPU se distingue d'abord par l'utilisation d'un ensemble limité mais bien choisi d'un certain nombre de fonctions mathématiques (sin, cos, puissance, la racine carrée, produit scalaire, etc.), ensuite par le procédé de calcul qui utilise naturellement les types vectoriels (de 3 ou 4 composantes) et finalement par l'utilisation courante des opérations en virgule flottante. Tout cela est encore majoré par la contrainte du temps dans le cas de la visualisation 3D interactive et/ou en temps réel. Ce calcul spécifique, réuni avec la masse d'informations traitées lors de ce calcul, était (et est toujours) inadapté aux architectures du calcul général, même si ces dernières auraient pu l'effectuer en dépit de la rapidité.

Puisque ni les GPP, ni leurs successeurs GPPMM, ne pouvaient satisfaire la demande importante pour la visualisation devenue de plus en plus exigeante, une nouvelle catégorie de processeurs dédiés a vu le jour, la catégorie des processeurs graphiques destinés à l'accélération du calcul pour la visualisation 3D. Ces processeurs étaient et sont toujours des accélérateurs, c'est-à-dire des processeurs secondaires dédiés, qui restent fortement liés aux processeurs principaux (GPP).

En effet, la cohabitation GPP-GPU a donné ce que l'on en avait espéré. On a séparé la problématique de la visualisation à deux parties : la partie de haut niveau est représentée par le GPP et on l'utilise pour l'exécution de l'application de la visualisation. La deuxième partie de niveau bas est représentée par ces accélérateurs dédiés et spécialisés qui s'occupent du calcul intensif. Toutes les architectures actuelles des ordinateurs personnels utilisent les accélérateurs graphiques ou, au moins, des sous-systèmes graphiques si on ne peut pas les classer dans la catégorie des accélérateurs au vrai sens du mot.

[1] La définition universelle et pertinente du mot *multimédia* est difficile à trouver, nous pouvons le définir comme *réunissant plusieurs et différents types de médias*, dans ce contexte des médias *électroniques*

[2] Remarquons que la locution *grand volume* est vague et contextuelle, elle signifie ici les unités, dizaines ou au maximum les centaines de Mo

Depuis leur naissance, les processeurs graphiques ont évolué et évoluent toujours très rapidement. Tout en profitant des innovations technologiques durant la dernière décennie, ils se sont développés à partir d'une structure du pipeline fixe capable d'exécuter seulement un ensemble très limité de fonctions jusqu'à un pipeline avec des blocs parallélisés et dotés de la possibilité de reconfiguration complexe. Cette faculté de reconfiguration de plusieurs de leurs composantes lors de l'exécution leur a permis de devenir une architecture très intéressante non seulement pour la visualisation 3D à laquelle ils étaient originalement destinés, mais également pour une utilisation plus large et moins liée à la visualisation qui inclut à présent aussi le calcul scientifique[GPG].

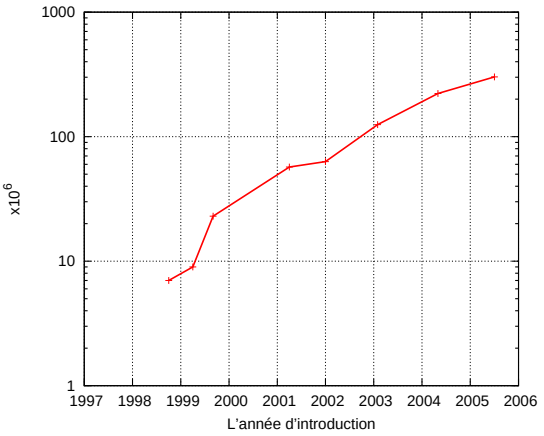


FIG. 1.2 : Évolution de nombre de transistors des processeurs graphiques NVidia

Pour établir un parallèle avec la Loi de Moore et les GPP, q.v. fig. 1.1, nous présentons ici et de la même manière l'évolution des performances des processeurs graphiques, q.v. fig. 1.2[1]. En consultant cette figure, nous essayons de saisir la tendance de l'évolution du nombre des transistors par GPU dans le temps et nous nous apercevons rapidement que nous pouvons proposer deux interprétations différentes de ce graphique : une première et encourageante qui, en prenant en compte toute la période observée de sept ans, estime l'évolution comme exponentielle. Mais nous pouvons proposer également une deuxième interprétation qui, en se basant sur les quatre dernières années, ne devrait pas manquer de mentionner une légère baisse de la croissance et devrait nous inciter à être prudent dans nos conclusions de prévision exponentielle. En même temps, le dernier échantillon, correspondant à Juin 2005, désigne le chip NVidia G70 (GeForce 7800 GTX) fabriqué par TSMC[TP05] en utilisant la lithographie 110 nm[NVi06]. On n'est pas donc dans le même esprit que pour la fig. 1.1 qui présentait la technologie Intel, plus avancée à présent, et nous pouvons donc espérer encore un important progrès à venir en ce qui concerne l'évolution des GPU.

Nous remarquons qu'en complétant les GPP dans le domaine dans lequel ils n'étaient pas très adaptés, les GPU ont pu explorer la voie de la forte parallélisation sur une seule puce. Leur prix de fabrication les a rendus très compétitifs par rapport aux autres solutions du calcul parallèle, notamment par rapport à la parallélisation via la multiplication du nombre de processeurs. La forte spécialisation a également son revers. Étant dédiés à un calcul spécifique, les GPU présentent de nombreuses particularités qui peuvent les rendre très utiles pour une tâche ou un type de calcul mais qui peuvent aussi les rendre complètement inadaptées pour d'autres. Nous allons discuter de ce sujet dans la partie *Algorithmes et les skeletons algorithmiques*.

Dans les applications pratiques, c'est surtout la performance qui nous intéresse. Si on voulait effectuer une prévision d'évolution des performances des architectures de la même façon que l'a fait Moore avec l'évolution du nombre des transistors sur une puce, nous trouverions que la transposition de cette idée sur la performance n'est pas triviale. La performance d'un système dépend non seulement du processeur principal mais également de toutes ses composantes qui doivent être bien équilibrées car c'est tout l'ensemble qui forme un seul produit dont la performance nous intéresse.

Même si les auteurs utilisent les courbes d'évolution des GFLOPS[OLG+05, Owe04] ou même d'autres paramètres de performance pour comparer les CPU avec les GPU afin de soutenir la supériorité des GPU pour le calcul, nous devons souligner qu'il est très difficile de comparer la performance de deux architectures très différentes qui, en plus, ont été conçues chacune pour un calcul distinct. Nous n'allons pas faire une telle comparaison et nous allons présenter l'évolution des paramètres de performances des GPU indépendamment des GPP.

[1] Les chiffres correspondent aux processeurs de NVidia Corporation et ont été collectés à partir des sources indirectes depuis l'internet

La figure 1.3[1] présente l'évolution des trois paramètres principaux d'un GPU – les capacités maximales de traitement en nombre de *vertex*[2] par seconde, en nombre de *texels*[3] par seconde et en nombre de *fragments*[4] par seconde. Différents graphiques de ce type sont présentés par différents auteurs [Owe04, Lue04, Mah05, Sei04]. Nous constatons que les courbes correspondant aux fragments et texels sont d'une forte croissance et suivent, au moins pour les quatre dernières années, une fonction exponentielle. L'exception est la courbe des vertex. Elle est également d'une forte croissance mais, pour les années 2003 à 2005, elle ne suis pas la même progression. Ceci est normal car les unités de traitement des vertex n'ont pas besoin du même degré de parallélisation que les texels et les fragments. Les unités de traitement des vertex sont conçues pour un nombre relativement petit de données en comparaison avec le nombre beaucoup plus important des fragments traités dans les unités parallélisées plus massivement. Le traitement des texels est effectué principalement dans les unités de traitement des fragments, c'est pourquoi les chiffres pour les texels coïncident avec ceux des fragments.

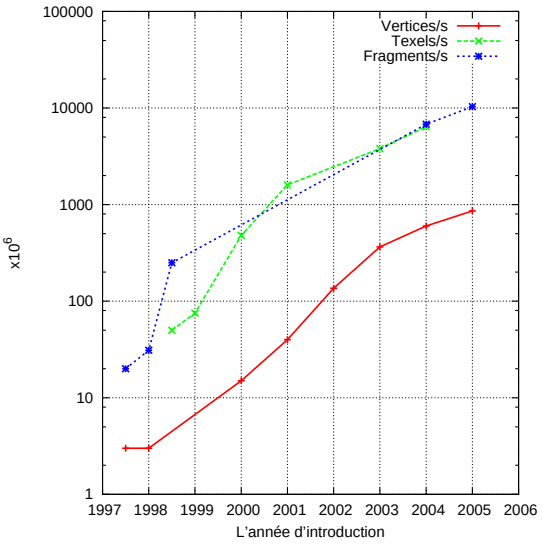


FIG. 1.3 : Évolution des performances des processeurs graphiques NVidia

Ce que nous voulions démonter c'était le fait que pour les GPU ce n'est pas seulement le nombre des transistors qui présente une croissance exponentielle mais que ce sont également les paramètres de performance. Ces croissances sont très encourageantes, elles suivent la logique de la Loi de Moore et nous laissent penser à la forte évolution des GPU qui nous attend encore dans les prochaines années.

1.4 Au-delà de l'horizon

Nous avons utilisé la leçon de l'histoire concernant la Loi de Moore pour bien souligner le dynamisme exponentiel d'évolution des performances des puces électroniques durant les 40 dernières années. Ce dynamisme n'a pas cessé depuis, au contraire, et nous pouvons déjà entrevoir les grands changements qui nous attendent dans les années à venir. Il est quasiment inutile de remarquer que ce que nous pouvons fabriquer aujourd'hui est loin de ce que pouvaient espérer les membres[5] de l'équipe de Bell Labs quand ils ont découvert l'effet de transistor en 1947. La question que nous pouvons nous poser est si dans 40 ans nos successeurs, eux aussi, considéreront notre époque de la même manière que nous regardons aujourd'hui l'époque de Moore. Nous pouvons nous demander si l'idée exprimée par Moore en 1965 pour le nombre des transistors par une seule puce électronique peut encore résister aux innovations technologiques et aux nouvelles tendances dans le design d'architectures. Moore lui-même la critiquait avec le résultat plutôt optimiste en 2003 dans son article [Moo03a, Moo03b] intitulé *No exponential is forever : But "Forever" Can Be Delayed !*

En effet, l'avenir proche est assez prévisible [FH05]. Comme[6] l'indique International Technology Road-

1 Les chiffres ont été collectés à partir des documents de communication de NVidia Corporation

2 Dans le contexte des GPU, un vertex contient des coordonnées d'un sommet d'une forme géométrique et éventuellement d'autres informations (couleurs, position dans la texture, etc.)

3 Abréviation du terme anglais *texture element*, texel, contient des informations d'un élément de texture

4 Dans le contexte des GPU, un fragment contient des coordonnées 2D de la position sur l'écran, la coordonnée z et l'information sur la(les) couleur(s)

5 John Bardeen, Walter Brattain et William Shockley ont découvert l'effet de transistor en décembre 1947 [Lab97] et ils ont obtenu le Prix Nobel de physique en 1956. Il est intéressant de s'apercevoir que le brevet [Sho51] issu de cette découverte n'a été déposé qu'au nom de William Shockley.

6 Selon la thèse de Robert Strzodka [Str04], pages 103–104

map for Semiconductors[ITR], le dynamisme de la croissance exponentielle pourrait ralentir dans les années à venir et être égal à 3 années pour le doublement du nombre des transistors. Il devrait persister sous cette forme pendant au moins 10 années. Autour de 2020, quand on atteindra les contraintes physiques de fonctionnement de la technologie basée sur les charges électriques, d'autres approches qui ne seraient pas limitées par ces contraintes pourraient rallonger la durée de la Loi de Moore.

En même temps, les approches parallèles commencent à se présenter comme un phénomène quotidien. Les processeurs existants avec plusieurs cœurs destinés aux applications parallèles et d'autres solutions variées qui suivent cette philosophie et sont étudiées par les chercheurs[1] en sont la preuve. Il est fort probable que dans un avenir très proche nous allons nous retrouver couramment aux prises avec les solutions du grand public composées de plusieurs processeurs physiques, eux-mêmes déjà dotés d'une structure parallèle à l'intérieur.

En ce qui concerne les GPU, nous pouvons constater qu'ils se sont transformés en processeurs programmables, très adaptés à certains types du calcul et qu'ils commencent à être sérieusement utilisés même pour les tâches qui ne leur étaient pas destinées à l'origine. Les chances pour une future utilisation plus massive des GPU dans l'avenir semblent très bonnes. Ce qui parle dans leur intérêt c'est la façon implicitement parallèle de programmation et d'exécution, propres à ces processeurs. Leur architecture est adaptée à ce travail et peut être facilement élargie au niveau de la puce sans contredire la philosophie de la structure.

Ce qui semble pour l'instant le vrai frein de l'utilisation vulgaire des GPU pour le calcul général est la manière dont ils sont incorporés à côté des GPP dans l'architecture PC. Remarquons surtout la mémoire distincte des GPU qui nécessite le transfert des données dans les deux directions et également la lenteur du pilote en temps de réponse à un grand nombre de petits blocs de commandes. Espérons que ces obstacles majeurs vont bientôt disparaître, que le calcul général sur les GPU pourra bientôt atteindre sa maturité et devenir courant dans son utilisation. Les premiers pionniers à montrer le chemin pourraient être les consoles de jeux. En se différenciant de la plateforme PC des ordinateurs grand public, elles combinent dans une architecture dédiée une grande puissance du calcul général avec une grande puissance du calcul spécialisé sur les flux de données. Pour ces consoles, nous pouvons nous attendre également à une forte évolution dans les prochaines années.

À ce jour, nous regardons vers l'avenir et nous imaginons et créons les solutions pour le futur. Soit à court terme et exploitables dès à présent ou très prochainement, soit à long terme, plus conceptuelles et exploitables dans quelques années. Prévoir avec exactitude ce qui se passera à long terme est très difficile. En revanche, nous pouvons dès aujourd'hui cibler les idées prometteuses et les concepts qui ont du potentiel. Nous croyons que les architectures que nous visons dans cette thèse et les algorithmes que nous y décrivons en font partie.

[1] Eva Dejnozkova a étudié cette approche dans sa thèse [Dej04] en se focalisant sur les processeurs à faible consommation

CHAPITRE 2

État de l'art

Cette thèse est consacrée aux algorithmes de la morphologie mathématique qui perçoivent et traitent les données comme flux. Nous parlons ainsi du *paradigme flux* ou également du *paradigme stream*. Précisons que le terme français *flux* et le terme technique d'origine anglaise *stream*, largement utilisé dans le domaine informatique, désignent le même sujet. Bien que nous utiliserons les deux dans les locutions désignant le traitement ou le paradigme, pour une distinction plus pointue nous préférons employer le terme *stream* comme type de données et le terme *flux* dans l'appellation d'une méthode de traitement.

Ce sujet est d'un intérêt majeur pour les réalisations des applications pratiques et plus particulièrement pour les applications en temps réel où les architectures du calcul sont souvent dédiées et présentent les caractéristiques de traitement en flux. En même temps, ce sujet n'est pas, selon nous, suffisamment décrit dans sa globalité. Même si certains articles publiés dans les journaux scientifiques traitent de sujets connexes à cette problématique et quelques thèses ont été également dédiées à la conception des architectures particulières pour la morphologie mathématique, nous n'avons pas connaissance d'un travail qui serait intégralement consacré aux techniques algorithmiques du domaine de la morphologie mathématique, dédiées aux architectures possédant des capacités spéciales pour le traitement en flux, y compris les techniques utilisant le pipeline graphique et les processeurs graphiques pour les fins de la morphologie mathématique.

L'ambition de chaque scientifique est, bien sûr, de pouvoir découvrir un domaine ou une partie d'un domaine qui serait complètement inexploré. Acquérir les expériences dans un tel domaine, en approfondir les connaissances et montrer par la suite le chemin à ses successeurs, c'est la vocation de chacun d'eux. Pourtant, il est très rare de nos jours que nous puissions explorer une problématique *hic sunt leones*[1], une problématique qui serait totalement inexplorée et dont nulle partie ne serait encore divulguée par d'autres chercheurs. C'est peut-être le phénomène de notre époque où nous travaillons à l'échelle planétaire et dans une atmosphère très compétitive où il est tout-à-fait possible que certaines équipes scientifiques travaillent sur les mêmes problématiques parallèlement car les demandes pour leurs études proviennent de différentes sources et sont assurées par un financement distinct. Ce qui nous arrive le plus souvent dans un tel environnement, c'est que nous explorons les problématiques intéressantes et peu connues, mais déjà prospectées par un tiers, les problématiques qui ne sont pas complètement vides mais dans lesquelles nous constatons un manque effectif d'une expertise élaborée préalablement.

Il faut remarquer que divers articles et travaux scientifiques décrivant l'utilisation des architectures SIMD pour le traitement d'images et le traitement par la morphologie mathématique sont présents et assez nombreux. En revanche, les travaux de ce domaine pour les processeurs graphiques sont quasi inexistants car il s'agit d'une problématique nouvelle qui n'a pas pu encore d'être étudiée en profondeur

[1] L'inscription latine *hic sunt leones*, signifiant en français *ici sont les lions*, était utilisée sur anciennes cartes pour désigner un espace inconnu

ni être utilisée à une large échelle. De plus, nous avons remarqué le manque de rigueur dans la description formelle de tels algorithmes. Le degré de formalisation varie d'un article à l'autre et ce fait est le plus flagrant dans les descriptions des algorithmes pour les processeurs graphiques où l'on se contente, de temps à autre, seulement d'une image d'illustration ou d'une description vague de fonctionnement. Dans ces situations, nous serions contents de pouvoir nous adresser à une approche formelle, simple, unifiée et surtout déjà existante pour la description des algorithmes pratiques de traitement d'images et cela non uniquement en morphologie mathématique.

C'est ainsi que nous percevions l'état de l'art au moment de notre intervention et d'où est née notre motivation de chercher à clarifier une partie de la morphologie mathématique concernant le traitement sur les architectures dites *orientées flux*. Au début, nos cibles prioritaires étaient les architectures grand public à basse consommation avec les fonctionnalités multimédia qui sont de plus en plus utilisées dans les applications embarquées de traitement des images. Mais nous ne voulions pas nous restreindre seulement aux architectures à basse consommation car le principe de travail avec ces dernières est le même que pour les architectures classiques grand public qui sont désignées par le terme anglais *main stream*. En même temps, le style de travail en flux de données est bien marquant dans le fonctionnement des processeurs graphiques et nous voulions démontrer également la piste de leur possible utilisation pour les applications de la morphologie mathématique.

Toutes ces architectures utilisent le paradigme stream comme le modèle de traitement. C'est la raison pourquoi nous voulions décrire les algorithmes de la morphologie mathématique qui seraient construits en utilisant l'approche des flux de données pour le traitement. Les algorithmes qui seraient en même temps assez généraux dans leur forme et qui pourraient ainsi rester abstraits d'une architecture particulière. Cette généralisation est, en effet, à l'origine de notre titre qui emploie explicitement le terme *architectures orientées flux* pour désigner un groupe d'architectures dont le point commun est le traitement en flux.

Nous voudrions présenter, dans la suite de ce petit chapitre, les sources qui nous ont inspiré et dont le travail nous a servi comme le point de départ pour nos propres études.

Skeletons algorithmiques et formalisme fonctionnel

En mathématique, il existe plusieurs manières de décrire un algorithme. Nous voudrions en choisir une qui permettrait de décrire d'une façon simple le travail avec les flux et qui serait à la fois abstraite, pour que les procédés que nous décrivons pour les architectures spécialisées soient exprimés le plus généralement possible, et qui aurait à la fois la possibilité de vérifier la bonne formulation de nos prescriptions.

C'est pourquoi nous avons éliminé tout au début l'éventualité d'utiliser le pseudocode [Wik05c] et nous nous sommes inclinés vers les méthodes utilisées pour la description et pour la programmation des procédés destinés aux architectures parallèles. Lors de nos recherches bibliographiques, nous nous sommes vivement intéressés aux approches des skeletons algorithmiques exprimés en formalisme fonctionnel. Ces travaux sont souvent originaires du Royaume Uni, largement représentés par des articles[1], des rapports techniques [DGG+96, Dar98] ou des présentations [DNG00] de Darlington et al. de *Departement of Computing* d'*Imperial College* à Londres mais aussi des thèses doctorales [To95, Yan97, Gha99] issues du même établissement. Un autre groupe scientifique qui travaille sur cette problématique est celui de l'Université de Pisa en Italie et dont nous citons un article antécédent [DMO+92] aux travaux de Darlington et al., une thèse doctorale [Pel93] et les travaux récents [AD03] issus du même centre de recherche. Il existe, en effet, des travaux encore plus anciens traitant le même sujet [Col89] mais également les travaux relativement récents et d'origine française qui traitent les aspects pratiques de la programmation pour les architectures parallèles [Cou02]. Les travaux touchant l'emploi du formalisme fonctionnel pour les architectures SIMD [Jou91] peuvent également être retrouvés.

1 Articles de Darlington et al. [DFH+93, DT95, DkGT95, DGTY95b, DGTY95a, DkGTY96, ADGkG96, DGGT97]

Les techniques utilisées pour dériver les programmes à partir des spécifications formelles par skeletons algorithmiques sont également présentées. Il s'agit des techniques connexes au formalisme de Bird et de Meertens qui ont proposé une algèbre pour la programmation et on parle de *Bird-Meertens Formalism*[Gib94] appelé également *Squiggol* ou *Squigol*. Cette algèbre, utilisée pour la programmation, peut être perçue comme un langage fonctionnel. D. B. Skillicorn[1] a évoqué[Ski92] l'utilisation possible de ce formalisme en tant que modèle parallèle pour la programmation.

Un article qui compare différentes approches à la programmation parallèle par skeletons algorithmiques et qui essaie de les classifier est présenté par Campbell[Cam96]. Tous ces travaux présentent des méthodologies qui nous paraissent propices à nos besoins. Il s'agit, en effet, des méthodologies complexes et il faut noter que nous n'en avons repris qu'une partie, mais une partie suffisante à nos explications pour le calcul en stream et les manières de sa parallélisation.

Morphologie mathématique appliquée

C'est l'utilisation pratique de la morphologie mathématique qui est visée par les approches que nous décrivons dans cette thèse. Lors de l'utilisation des algorithmes de traitement d'images par la morphologie sur les architectures grand public, nous nous sommes aperçus que les implémentations des méthodes de base que l'on utilisait pour la construction des algorithmes complexes de traitement d'images se consacraient prioritairement à la modularité des opérateurs et étaient principalement destinées à un travail scientifique en phase d'études de faisabilité plutôt qu'à la phase applicative des projets industriels de traitement d'images en temps réel.

C'est, en effet, ce type de projets industriels qui pose, parmi d'autres, une contrainte sur la durée maximale d'exécution d'un tel traitement. Une telle contrainte n'est pas présente dans la plupart des travaux de recherche effectués pendant la phase de construction des prototypes des algorithmes de traitement d'images. Ces travaux de recherche non appliquée peuvent, en effet, aboutir à des temps du calcul beaucoup plus élevés, excluant même les algorithmes résultant du fonctionnement en temps réel.

Pourtant, les applications industrielles de traitements d'images pour le temps réel qui utilisent les méthodes de la morphologie mathématique et font appel aux ordinateurs grand public, soit classiques, soit à basse consommation, deviennent de plus en plus fréquentes. Ceci surtout grâce à l'augmentation de leurs performances durant les dernières années, ce qui leur a permis de devenir compétitifs par rapport aux architectures particulières et entièrement dédiées utilisées le plus souvent dans les applications de la la morphologie mathématique pour le temps réel.

De ce point de vue, c'est la thèse doctorale de Cristina Gomila[Gom01] qui nous a inspiré car elle présente un algorithme intéressant qui combine le filtrage morphologique de bas niveau avec la segmentation d'images et le suivi des objets dans le traitement des données vidéo. Dans la partie pratique de sa thèse, Gomila vise l'utilisation de cet algorithme dans les applications de téléphonie mobile sur le matériel embarqué grand public.

En effet, ce sont les applications de ce type que nous visons pour les algorithmes spécifiques décrits dans cette thèse comme des applications potentielles de la morphologie mathématique destinées aux architectures multimédia grand public.

Architectures dédiées et algorithmes relatifs

Vu la nature fortement parallèle et itérative des méthodes morphologiques de traitement d'images, il n'est pas surprenant que ces dernières constituent un centre d'intérêt privilégié des constructeurs des architectures particulières de traitement d'images et des concepteurs des algorithmes pour les architectures parallèles. Cet intérêt est présent depuis l'apparition de ces méthodes et depuis leurs utilisations en pratique pour les applications où le temps de traitement est un facteur limitant majeur.

[1] Skillicorn est aussi l'auteur d'une taxonomie complexe pour la classification des architectures[Ski98]

C'est pourquoi les chercheurs d'aujourd'hui ont déjà à disposition un certain nombre de travaux qui décrivent différentes réalisations du matériel informatique spécialisé en traitement de la morphologie mathématique et un certain nombre de travaux portant sur les aspects algorithmiques de traitement d'images destiné aux architectures parallèles.

Parmi les travaux qui sont consacrés aux algorithmes parallèles pour la morphologie mathématique, nous mentionnons les articles[MR96, RM00] et la thèse doctorale[Mei04] d'Arnold Meijster. Nous pouvons citer également les travaux d'Andreas Bieniek et al.[LBB98, BM98, BBM+96] qui exploitent diverses possibilités de construction des algorithmes de segmentation des images pour les architectures parallèles. Les algorithmes de la transformation distance et leur utilisation pratique pour les applications médicales sont décrits dans la thèse doctorale[Cui99] d'Olivier Cuisenaire.

Les architectures dédiées à certaines méthodes de traitement de la morphologie mathématique et les algorithmes conçus spécifiquement pour ces architectures sont représentés par les implémentations des automates cellulaires à l'échelle des pixels. Le traitement en flux de données est représenté le plus souvent par diverses architectures optimisées pour les applications de la morphologie travaillant avec des signaux vidéo. Ce sont les lignes en retard suivant le sens de balayage vidéo qui sont utilisées comme paradigme pour l'exploitation de la localité et le parallélisme de données lors du travail avec le voisinage proche. Citons les travaux traitant ce sujet – un article[Nog97] de Dominique Noguet et sa thèse[Nog98] doctorale mais également la thèse de Fabrice Lemonnier[Lem96] et la thèse de Raphaël Sasportas[Sas02], ces deux dernières issues du Centre de Morphologie Mathématique.

Une autre groupe d'architectures parallèles est représenté par les travaux[DD04, DD03] d'Eva Dejnozkova et al. qui décrit dans sa thèse[Dej04] une architecture convenable pour les algorithmes traitant les images par les équations partielles différentielles.

La construction de la partie d'équipement logiciel pour une architecture dédiée à la morphologie mathématique peut trouver également l'inspiration dans la thèse doctorale[Bil92] de Michel Bilodeau qui traite ce sujet.

Traitement d'images sur les processeurs graphiques

Les processeurs graphiques ont déjà été utilisés pour les calculs des opérations de la morphologie mathématique. L'article le plus ancien (année 2000) que nous avons pu consulter qui est consacré à ce sujet est celui de Hopf et Ertl[HE00] et utilise la construction des opérations morphologiques par un rendu consécutif des rectangles décalés affichant l'image stockée en tant que texture et par leur fusion consécutive dans le framebuffer utilisant les opérations max et min pour le *blending*.

Plus récemment, en 2002, Yang et Welch ont publié un article[YW02] où ils décrivent l'utilisation du hardware graphique grand public pour l'extraction d'un objet de la scène par différence avec l'image de fond. Ils utilisent par la suite le post-traitement par ouvertures morphologiques. Les opérations morphologiques y sont implémentées à l'aide des possibilités du *multitexturing* du processeur graphique qu'ils ont eu à disposition ; ce qui distancie celle-ci de celle présentée par Hopf et Ertl mentionnée ci-dessus.

Récemment (2005), Dixit, Keriven et Paragios ont décrit la technique pour la segmentation des images basée sur les graphes et le *Maximum flux problem*, et utilise l'algorithme *push-relabel* pour le solutionner. Ils ont développé une implémentation de cet algorithme destiné aux processeurs graphiques. Le calcul principal est effectué dans les unités de traitement des fragments où leur algorithme bénéficie implicitement d'une parallélisation du calcul.

Le sujet ne traitant pas directement la morphologie mathématique mais qui est crucial pour les applications de traitement d'images est celui de l'utilisation des processeurs graphiques pour les algorithmes basés sur les équations partielles différentielles. Ce sujet est bien expliqué dans les articles de Strzodka[Str02] ou dans les articles que Strzodka a publié avec Rumpf[RS01] ou encore dans les articles où ils sont cités tous les deux comme les coauteurs[DPRS01, SDR04, KEH+02]. La problématique des équations partielles différentielles utilisées en traitement d'images est bien décrite dans la thèse doctorale de Keriven[Ker97] ou de Strzodka[Str04]. Ce dernier a également participé à la publication d'un article[ST04] traitant de

l'implémentation de la fonction distance généralisée et de la skeletonisation dans le hardware graphique. Celle-ci étant basée sur les pixels des contours d'un objet, elle utilise le pavage par la décomposition hiérarchique et est particulièrement intéressante pour les applications où un objet couvre une partie relativement grande de l'image, e.g. pour déterminer les cartes de distance pour la navigation des robots.

Wikipedia

De temps à autre, nous faisons appel dans nos explications à Wikipedia[Wik06k] qui est un projet d'encyclopédie libre et gratuite et dont les articles nous paraissent suffisamment explicites pour que nous puissions les proposer au lecteur comme une source bibliographique rapide pour davantage d'informations ou de précisions sur certains points. Notamment pour donner, d'une façon claire, une définition aux termes qui sont largement employés mais dont le sens reste assez vague, citons comme exemple le terme *multimédia*[Wik06h].

Cette page est blanche par intention

CHAPITRE 3

Architectures

3.1 Taxonomie des architectures

Pour pouvoir comprendre dans quel domaine se placent les algorithmes décrits dans les parties suivantes, il nous faut spécifier la classe des architectures du calcul appropriées. C'est pourquoi nous allons présenter les taxonomies – les systèmes de classification – des architectures parallèles.

Les différenciations suivantes gardent une très bonne généralisation par rapport au fonctionnement exact des machines parallèles ce qui nous permettra d'obtenir une protabilité entre différentes implémentations sur des architectures précises, soit existantes, soit des architectures à concevoir, et élargir ainsi le champ d'applications possibles de nos algorithmes.

Il existe plusieurs systèmes de classification des architectures existantes. Pour une liste exhaustive des travaux portant sur la caractérisation des architectures, parallèles ou séquentielles, nous recommandons un article comparatif[BD93] des différentes taxonomies.

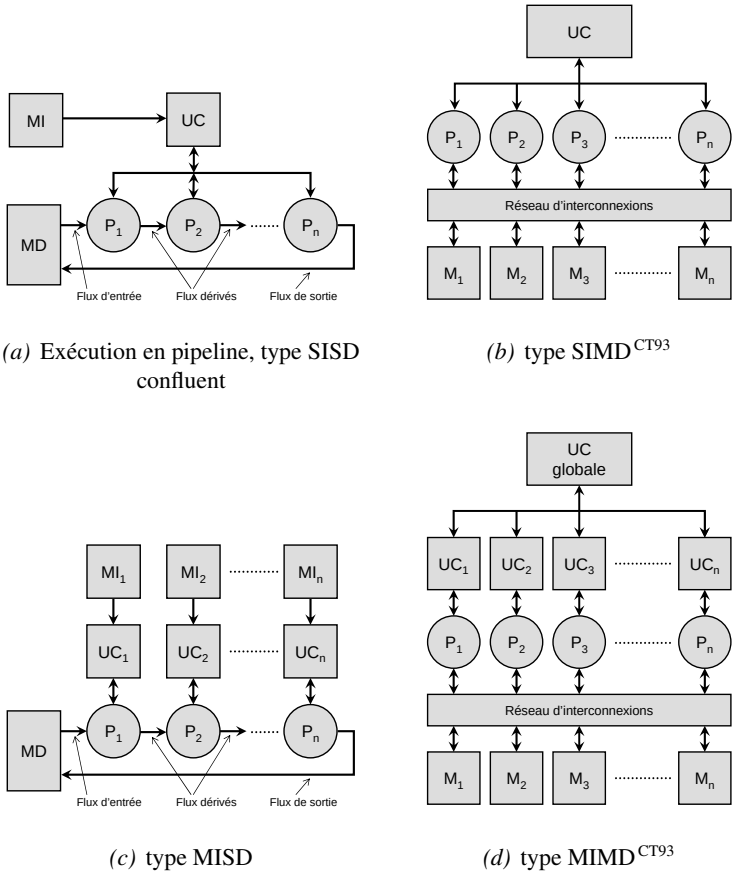
3.1.1 Taxonomie de Flynn

Parmi les diverses façons de caractériser les architectures, celle de Flynn[Fly66] détient une place privilégiée car elle est toujours perçue comme une façon simple et presque intuitive de la description du mode de fonctionnement d'une machine. Elle caractérise les machines selon le type de flux d'instructions et de flux de données.

En dépit de cette simplicité, cette taxonomie n'est pas l'une des plus discriminatoires et présente ainsi de grands désavantages[CT93], principalement dues à l'absence de caractérisation du système de la mémoire. Malgré les travaux[Dun90, Ski98] essayant de compléter ce système de caractérisation, la taxonomie de Flynn reste toujours le premier outil de description dans un grand nombre de publications scientifiques. Nous la mentionnons pour pouvoir introduire et expliquer les termes qui seront utilisés par la suite.

La Taxonomie de Flynn classe les architectures en ces 4 catégories :

- architectures SISD et SISD confluant (Single Instruction (stream), Single Data (stream)) - à flux simple d'instructions et à flux simple de données,
- architectures SIMD (Single Instruction (stream), Multiple Data (stream)) - à flux simple d'instructions et à flux multiple de données,
- architectures MISD (Multiple Instruction (stream), Single Data (stream)) - à flux multiple d'instructions et à flux simple de données,
- architectures MIMD (Multiple Instruction (stream), Multiple Data) - à flux multiple d'instructions et à flux multiple de données.



(a) Exécution en pipeline, type SISD confluent

(b) type SIMD [CT93]

(c) type MISD

(d) type MIMD [CT93]

FIG. 3.1 : Les types d'architectures selon la taxonomie de Flynn. Légende : UC - unité centrale, P - processeur, M - mémoire, MI - mémoire d'instructions, MD - mémoire de données

3.1.1.1 Architecture SISD et SISD confluant

Dans ce type d'architectures, nous pouvons distinguer le mécanisme sériel de décodage d'instructions et, par conséquent, le traitement d'un seul flux d'instructions. Une seule donnée peut être traitée dans le laps de temps Δt, dérivé du cycle d'horloge et lié ainsi à la fréquence de l'architecture.

Le type d'architectures SISD a un fonctionnement séquentiel et n'est pas, en effet, explicitement parallèle. Mais il englobe également le traitement SISD confluant et inclut ainsi les architectures avec un pipeline dont la structure est illustrée sur la fig 3.1(a).

Un pipeline est utilisé pour maximiser l'usage des blocs d'exécution en divisant une opération d'une granularité donnée en une séquence sérielle des opérations de granularité plus fine. Le traitement est toujours effectué avec un seul flux d'instructions sur un seul flux de données. Seule une donnée peut être traitée par une unité de traitement distincte dans l'intervalle Δt. Avec cette structure, l'unité du calcul parvient à une augmentation de bande passante exprimée par le nombre d'instructions par seconde. Nous présentons un exemple pratique d'exécution en pipeline du processeur SH-5, q.v. fig 3.2.

3.1.1.2 Architecture SIMD

Dans ce type d'architectures, cf. fig 3.1(b), un seul flux d'instruction est utilisé pour contrôler le fonctionnement de plusieurs unités élémentaires du calcul. Ainsi, l'architecture est capable de traiter plusieurs données à la fois. Les données y arrivent comme un flux multiple, composé d'un certain nombre d'éléments de base. On classe dans cette catégorie également les machines vectorielles, les réseaux cellulaires et les réseaux systoliques.

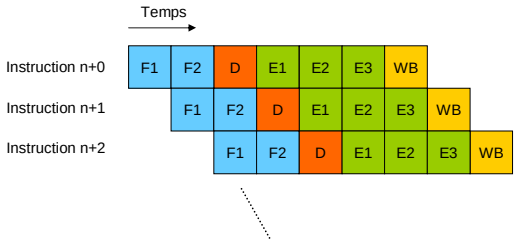


FIG. 3.2 : Exécution dans le pipeline du processeur SH-5 se poursuit de gauche à droite. Légende : F1, F2 (*fetch*) - phases de la mise en pipeline d'une instruction ; D - décodage de l'instruction ; E1, E2, E3 (execute) - jusqu'à 3 phases d'exécution, fonction exacte (mémoire, calcul en entiers, en virgule flottante) dépend de l'instruction, WB (*writeback*) - écriture des résultats dans le registre

Plus généralement, nous pouvons parler dans le même cadre d'une architecture SPMD (Single Program, Multiple Data) - architecture à un programme et à données multiples - qui a la même configuration mais les unités élémentaires du calcul exécutent d'une façon synchrone une même séquence d'instructions ou un même programme plus complexe. La façon synchrone d'exécution est ici indispensable et constitue le lien avec la catégorie SIMD travaillant de la même façon au niveau d'instructions.

3.1.1.3 Architecture MISD

Dans ce type d'architectures, plusieurs instructions sont destinées à traiter la même donnée. Chaque unité reçoit une instruction spécifique.

Ce type inclut également le traitement en pipeline où la tâche est divisée en étapes distinctes et où nous ne pouvons pas identifier plusieurs unités du calcul qui seraient indépendantes, une dédiée à chaque étape. En contraste avec le pipeline classé sous SISD confluent où les blocs du pipeline étaient commandés tous par une même unité centrale et où ils formaient un seul macro-bloc fonctionnel, le pipeline du type MISD correspond à un enchaînement des macro-blocs, chacun ayant sa propre unité centrale, cf. 3.1(c).

3.1.1.4 Architecture MIMD

La catégorie MIMD est implicitement perçue comme asynchrone ce qui la diffèrencie grandement d'autres catégories de la taxonomie de Flynn.

Les architectures de ce type sont composées de plusieurs unités de calcul séparées qui peuvent exécuter différents flux d'instructions sur différents flux de données, indépendamment les unes des autres et en utilisant leurs propres données locales, cf. 3.1(d).

Cette catégorie n'exige explicitement aucune forme de synchronisation entre les unités. Dans la pratique, la synchronisation, si on en a besoin, est effectuée par l'intermédiaire d'une mémoire partagée ou par le biais de passage des messages par un réseau d'interconnexions.

3.1.2 Taxonomie de Duncan

La taxonomie de Duncan[Dun90] est plus récente (1990) que celle de Flynn (1966). Elle était conçue principalement pour surpasser le manque de précision de la taxonomie de Flynn et ainsi pouvoir distinguer certains types d'architectures très proches dans leur fonctionnement.

La taxonomie de Duncan est basée sur la taxonomie de Flynn et reprend sa terminologie en incorporant les classes SIMD et MIMD dans un système plus discriminatoire et hiérarchisé. De plus, elle exclut les classes qui, selon Duncan, ne décrivent que les mécanismes parallèles de bas niveau. Ces mécanismes sont courants dans les structures des architectures contemporaines mais nous ne pouvons pas les classer rigoureusement comme parallèles car si tel était le cas, la plupart des ordinateurs modernes appartien-

draient aux architectures parallèles. Cette exclusion se présente par la non incorporation des classes SISD et MISD et par l'exclusion explicite des architectures ayant un pipeline au niveau des instructions.

En revanche, elle inclut les architectures vers lesquelles se porte notre attention. Il s'agit des architectures vectorielles, systoliques, des architectures array à vague et des architectures à flux de données. La figure 3.3 présente les catégories de cette taxonomie comme un arbre hiérarchisé.

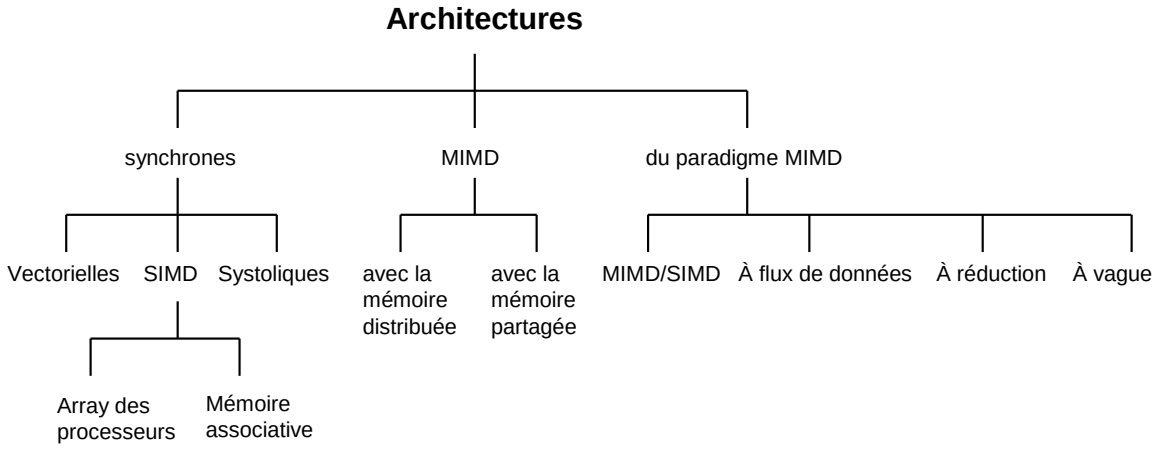


FIG. 3.3 : La taxonomie de Duncan

3.1.2.1 Architectures vectorielles

Les architectures de ce type sont caractérisées par plusieurs unités du calcul qui peuvent être chaînées et constituer ainsi un pipeline vectoriel. Leur grande différence par rapport aux architectures SIMD est le fait qu'elles sont conçues explicitement pour le calcul vectoriel et implémentent les fonctions spécifiques de ce calcul. Pour démontrer cette différence, nous pouvons citer l'exemple des instructions opérant entre les éléments d'un vecteur telles que la somme des éléments et qui ne sont pas en cohérence avec le modèle SIMD qui utilise l'application d'une seule et même instruction sur plusieurs unités et en principe indépendamment l'une de l'autre.

3.1.2.2 Architectures systoliques

Les architectures systoliques ont mérité également une place distincte dans la catégorie des architectures synchrones. Elles se présentent par un certain nombre d'unités capables d'exécuter d'une façon synchrone soit une instruction, soit, et le plus souvent, une séquence ou un programme. La présence d'un réseau d'interconnexions dont la topologie est prédéfinie par le type de calcul et la présence du phénomène de pulsation ou de pompage lors du transfert des données d'une unité du calcul à l'autre sont d'autres caractéristiques propres aux architectures systoliques.

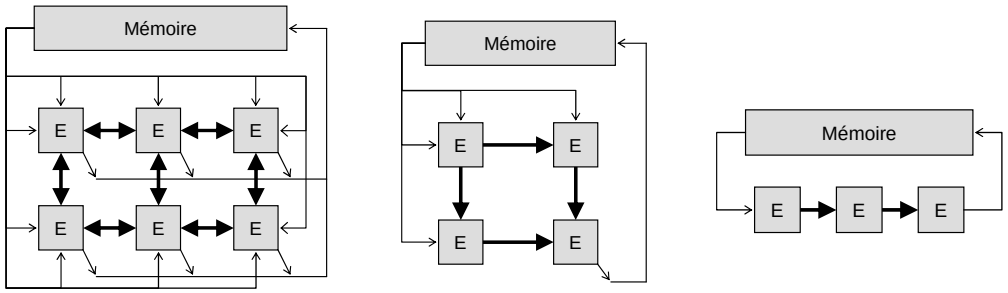
Les architectures de ce type gagnent leur performance en éliminant les goulots d'étranglement issus de l'accès aux données dans la mémoire. Plutôt qu'être écrits dans la mémoire principale, les résultats du calcul issus de chaque unité exécutive sont stockés dans leurs registres internes et sont distribués, dans la phase suivante du calcul, via le réseau d'interconnexion directement vers les unités qui en auront besoin pour la poursuite du calcul.

Cette catégorie est très proche de la catégorie très générale MISD de la taxonomie de Flynn, cf. 3.1.1.3, page 33. Les architectures systoliques se spécifient, en contraste de MISD, par leur insistance sur l'existence d'un réseau d'interconnexions complexes et sur la propagation des résultats à l'aide de ce dernier.

La structure systolique la plus souvent utilisée dans le traitement d'images est celle d'un array 2D des processeurs avec les interconnexions bidirectionnelles qui reflètent la définition du voisinage, cf. fig. 3.4(a). Nous pouvons en citer une réalisation intéressante destinée au traitement d'images qui intègre

une architecture systolique directement sur la puce du capteur CMOS et dont les éléments exécutifs sont dotés des fonctionnalités SIMD[GCRW+99].

Mais ce n'est qu'une des possibilités, un autre exemple de la configuration systolique avec la topologie adaptée pour la multiplication des matrices 2x2[Dun90] est présenté sur la fig. 3.4(b) et un exemple trivial d'une architecture systolique est un pipeline linéaire travaillant d'une façon synchrone et constitué des blocs fonctionnels distincts, q.v. fig. 3.4(c).



(a) pour le traitement d'images, grille carrée, 4 voisins

(b) pour la multiplication des matrices 2x2

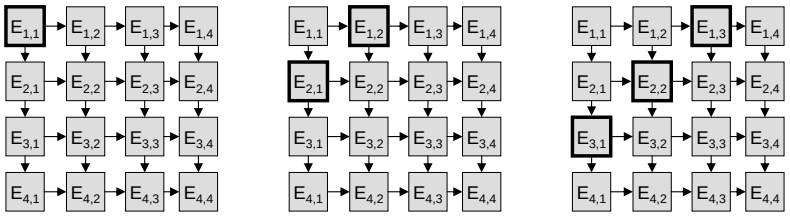
(c) pour le traitement en pipeline

FIG. 3.4 : Exemples des configurations et de la topologie d'interconnexions des architectures systoliques. Le réseau d'interconnexions est décrit par les flèches épaisses, les entrées et les sorties vers la mémoire sont décrites par les flèches fines. Légende : E - élément du calcul

3.1.2.3 Architectures array à vague

Les architectures array qui n'utilisent pas en même temps tous les éléments pour le calcul et n'en stimulent qu'un certain nombre sont nommées *architectures à vague*. Comme c'était le cas pour les architectures systoliques desquelles elles sont très proches, les architectures à vague sont composées d'un certain nombre de processeurs et d'un réseau d'interconnexions. Mais elles diffèrent de ces dernières par la façon asynchrone de passage des données par le réseau d'interconnexions. La donnée résultante d'un élément du calcul est passée à travers le réseau seulement si son successeur est prêt pour travailler avec cette donnée et la demande. On distingue ainsi une communication entre les éléments, ce qui n'était pas le cas chez les architectures systoliques où seule la donnée était transférée à travers le réseau.

La figure 3.5 illustre le fonctionnement d'une architecture à vague sur trois itérations. Différentes unités du calcul sont activées dans chacune des étapes, la forme précise de cette activation dépend du calcul à effectuer.



(a) itération no. 1

(b) itération no. 2

(c) itération no. 3

FIG. 3.5 : Exemple de fonctionnement d'une architecture à vague. Lors des trois itérations décrites, différents éléments (E) sont activés et effectuent le calcul. L'activation est illustrée par une bordure épaisse.

3.1.2.4 Architectures à flux de données

Les architectures *à flux de données* sont basées sur la perception des données en tant que flux et sur le passage des données d'une unité du calcul à l'autre. La caractéristique cruciale est la possibilité du lancement immédiat d'une instruction ou d'un programme dès que leurs opérandes sont disponibles. Il s'agit des architectures asynchrones dont la topologie du réseau d'interconnexions est dictée par le calcul à effectuer. Les unités exécutives sont spécialisées et ne s'occupent pas du va-et-vient des données pour le traitement car leur passage est déterminé à l'avance par le réseau d'interconnexions. La séquence d'exécution est basée sur les dépendances entre les unités. Exploitant cette dépendance, ces architectures peuvent implémenter tous les types du parallélisme, que ce soit au niveau des instructions, des routines ou des tâches.

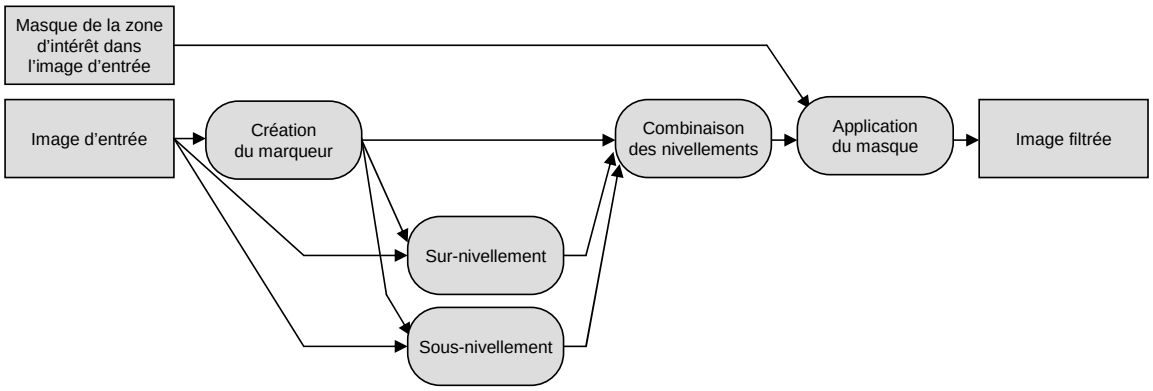


FIG. 3.6 : Exemple d'un graphe d'exécution d'un algorithme de filtrage morphologique qui utilise les nivellements. Les données transmises par le réseau d'interconnexions sont les images entières.

Nous présentons un exemple[1] d'un graphe d'exécution de l'algorithme de filtrage morphologique des images en utilisant les nivellements et le masque d'intérêt, q.v. fig. 3.6. Ce graphe décrit directement la structure d'une architecture à flux de données qui pourrait être envisagée pour ce traitement. Dans ce cas précis, les blocs exécutifs, représentés par les ellipses, sont des routines qui implémentent les opérations morphologiques et les données transmises le long des crêtes du graphe sont les images entières.

3.2 Facteurs influant la performance

Les algorithmes présentés plus loin dans cette thèse décrivent des procédés spécialisés mais leurs définitions ont été généralisées tant que les modalités exactes de leur exécution ou les détails précis d'implémentation ne sont a priori pas donnés. Pourtant, la réalisation de ces algorithmes peut faire appel aux architectures qui ont des structures différentes et qui utilisent les fonctionnalités matérielles particulières. Regardons maintenant quels sont les facteurs majeurs qui influencent la performance finale d'un système ou d'une architecture informatique et qui sont relatifs à l'implémentation matérielle. Par la suite, nous discutons en détail des points suivants :

- Structure de l'architecture dans 3.2.1,
- Fréquence de(s) l'unité(s) de calcul dans 3.2.2,
- Structure, capacité et fréquence des mémoires dans 3.2.3,
- Parallélisation des données dans 3.2.4,
- Parallélisation d'exécution dans 3.2.5,
- Écartement et latence des instructions dans 3.2.6,
- Instructions spécialisées dans 3.2.7,
- Nombre de registres et leur désignation dans 3.2.8,
- Approche à l'implémentation des algorithmes dans 3.2.9,

[1] Cristina Gomila [Gom01] utilisait les traitements de ce type dans sa thèse

3.2.1 Structure de l'architecture

La structure de l'architecture est cruciale et prédestine la performance finale et les scénarios d'utilisation d'un processeur ou d'un ordinateur issus de cette architecture.

La structure la mieux adaptée pour notre cas serait celle qui implémenterait nos algorithmes directement au niveau du matériel. Cette approche est courante dans la recherche et dans l'industrie spécialisée. C'est une démarche excellente pour la production et l'utilisation en masse. En revanche, elle peut s'avérer coûteuse pour les petites séries de produits, surtout à cause de la présence d'un long processus de développement et à cause d'un coût élevé de fabrication d'une série limitée. Les représentants apparents de cette approche sont les puces informatiques dédiées à une fin précise, les ASIC ou les architectures développées utilisant les FPGA.

Si on abandonne l'idée d'une architecture entièrement dédiée et si nous acceptons d'implémenter nos algorithmes sur des structures plus généralistes mais qui resteraient spéciales dans leur fonctionnement, on se retrouvera avec des architectures dédiées pleinement à un calcul sur les flux de données comme *Imagine*[1]. Ce type d'architectures peut constituer un bon compromis entre la performance et le coût de développement d'une application. Nous pouvons intégrer dans ce même groupe les processeurs graphiques programmables car malgré leur universalité marquée par la capacité de leur configuration lors d'exécution, leur architecture reste toujours dédiée.

De la façon la plus générique, nous parlons des architectures des processeurs généraux, des GPP. Pour notre travail – avec les flux de données – nous allons nous intéresser plutôt à leurs variantes ayant des capacités multimédia, GPPMM. Les avantages et les inconvénients sont intelligibles. Le prix est faible mais puisque la structure n'est pas nécessairement adaptée au travail que nous voudrions effectuer, la performance peut s'avérer inadaptée ou insuffisante.

Nous ne voudrions pas manquer de citer ici les exemples de structures très particulières que sont les consoles de jeux telles que Microsoft Xbox 360 [Wik06l] ou Sony PlayStation 3 [Wik06i]. Leurs structures sont dédiées à un but précis – visualisation interactive avec la sortie d'un flux vidéo. Elles combinent ainsi une partie largement spécialisée pour le travail sur les flux de données avec une partie générale qui assure le fonctionnement du moteur de l'application. Diverses applications sont possibles, e.g. un cluster du calcul scientifique a été construit [Cas03] à partir des consoles Sony PlayStation 2.

3.2.2 Fréquence de(s) l'unité(s) de calcul

La fréquence de l'unité du calcul est un paramètre cardinal car directement lié au débit d'exécution des instructions. Il est évident que si on augmentait la fréquence des unités du calcul, nous devrions nous attendre à une augmentation directe de la performance car nous allions, en théorie, traiter plus d'instructions par unité de temps. En pratique, il ne suffit pas d'augmenter uniquement la fréquence des unités du calcul mais nous devons ajuster aussi la fréquence de tous les systèmes subordonnés, notamment de la mémoire.

L'augmentation de la performance à travers augmentation de la fréquence des unités du calcul a été longtemps pourchassée et il faut noter que le défi de la conquête de la fréquence grandissante est toujours bien présent. Un autre point à souligner est le fait que la fréquence est directement liée, avec le voltage et la technologie de gravure, à la consommation et à la dissipation thermique. C'est une des raisons pour lesquelles nous ne voyons pas à présent de processeurs à faible consommation (moins de 10 W) et dont la fréquence dépasserait 500 MHz[2]. Pourtant, la fréquence ne devrait pas être la limitation principale car de nos jours on fabrique d'une façon industrielle les puces à 4 GHz. En revanche, ce que nous pouvons observer dans le secteur des puces à faible consommation c'est l'inclinaison vers les architectures VLIW[3]qui fonctionnent à une fréquence moindre mais qui augmentent la performance en multipliant les unités du calcul. Nous reparlerons de la consommation d'énergie dans la section 3.3.

1 Imagine Stream Processor [KDK+01, ORK+02, KDR+02]

2 Avec une exception : le processeur VLIW Transmeta Efficeon consommerait [Kra04] 3W à 1 GHz

3 réalisées par STMicroelectronics [Bag02, FBF00] et par Transmeta [Tra05b, Tra05a]

3.2.3 Structure, capacité et fréquence des mémoires

Indispensable pour le traitement, l'accès rapide à la mémoire est aussi important que la performance de l'unité centrale. Dans la plupart des cas aujourd'hui, les mémoires sont hiérarchisées selon la capacité et selon le temps d'accès qui est lié à la fréquence de la mémoire. Cette hiérarchisation est généralement dictée par le prix du produit. On se trouve ainsi avec un système composé de la mémoire principale et un certain nombre (1, 2, 3, voir plus) de mémoires dites caches qui sont plus rapides que la mémoire principale et dont la rapidité diminue et dont la capacité augmente à mesure que l'on s'éloigne de l'unité exécutive. Ces mémoires sont utilisées pour stocker les copies de travail des données. Ainsi, l'accès aux données durant le calcul est rendu plus rapide. Pour plus de précisions, notamment sur les types et le fonctionnement détaillé de la mémoire cache, nous adressons le lecteur à la littérature [LM99, Str04].

Idéalement, les données sont présentes dans la mémoire la plus rapide au moment où on en a besoin pour le calcul. Si tel était le cas, nous aurions une situation optimale. Pour y arriver et placer la donnée de la mémoire principale à la mémoire cache la plus rapide, diverses stratégies sont appliquées. On parle du *préchargement de la donnée* (angl. prefetch). Ce préchargement est implémenté soit au niveau du matériel et on parle ainsi du *préchargement automatique de données*, soit au niveau du logiciel en utilisant les instructions spécialisées dédiées au travail avec la mémoire cache dans le cas où notre matériel ne dispose pas du préchargement automatique. Tel est le cas, par exemple, pour le travail avec le processeur SH-5 [BHM+00].

Car, dans le cas où nous n'aurions pas effectué manuellement le préchargement et où la donnée ne serait pas présente dans la mémoire la plus rapide, l'architecture procéderait automatiquement au mécanisme du chargement instantané ce qui aurait ralenti ou stoppé l'exécution du programme jusqu'à ce que la donnée ait été transférée dans cette mémoire. Et ce qui causerait, bien sûr, de graves implications sur la performance.

Nous présentons un exemple pratique de cette situation pour le processeur SH-5, q.v. fig. 3.7. À remarquer notamment la façon d'exécution des instructions *LD.Q* et *LDX.Q* (cf. moitié supérieure de

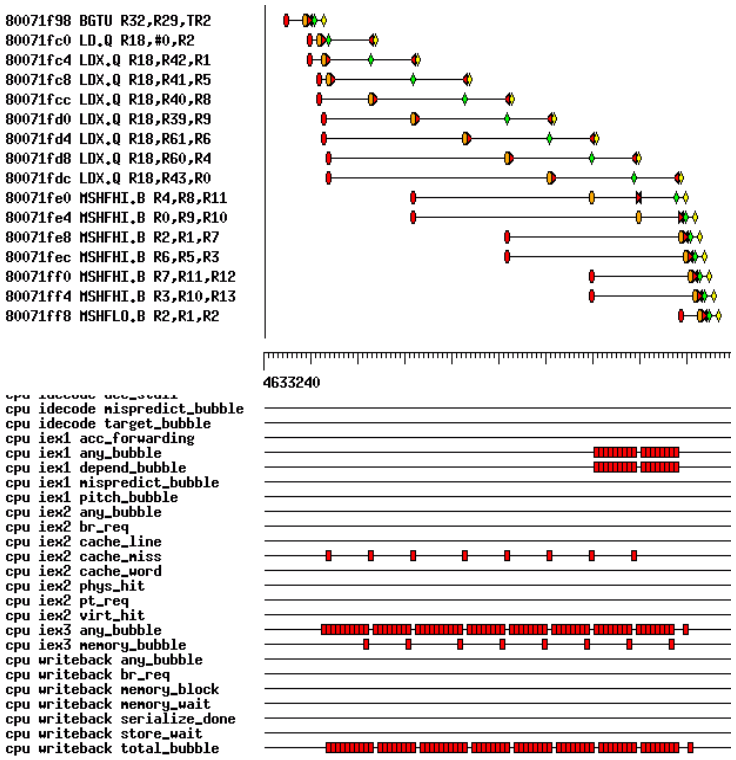


FIG. 3.7 : État du pipeline du processeur SH-5 lors de l'exécution dans le cas de données non présentes dans la mémoire cache

la fig. 3.7) qui, ne pouvant pas accéder aux données dans la mémoire cache (cf. *cache miss* dans la moitié inférieure de la fig. 3.7), déclenchent le chargement des données à partir de la mémoire principale. Pendant la durée de ce chargement (cf. *total bubble* dans la moitié inférieure de la fig. 3.7), le processeur ne peut pas poursuivre l'exécution et le temps est perdu inutilement avec une grande répercussion sur la performance – ici 10 cycles supplémentaires pour chaque instruction *LD.Q* ou *LDX.Q*. Pour comparer : si les donnés étaient présentes dans la mémoire cache, ces instructions auraient pris 1 cycle chacune.

3.2.4 Parallélisation des données

Une catégorie de méthodes dans le traitement des images, incluant la morphologie mathématique, sont les traitements sur les données ayant le parallélisme implicite. Cette propriété, si elle est bien exploitée, peut rendre le calcul plus rapide et cela d'une façon importante.

C'est, en effet, la manière la plus simple d'augmentation de la performance. Plutôt que de traiter une donnée à la fois, nous multiplions nos moyens matériels pour en traiter plusieurs en même temps. La manière précise de cette multiplication peut différer. Si on parle de traitements SIMD, nous utiliserons la même instruction pour commander d'une façon synchrone plusieurs unités de calcul qui peuvent traiter ainsi plusieurs données du même type. Si on parle de la parallélisation au niveau des blocs capables d'exécuter chacun une procédure ou un programme d'une façon asynchrone, nous allons nous référer le plus souvent aux traitements MIMD.

Les données utilisées dans la morphologie mathématique sont de ce type. S'il s'agit d'un signal 1D, d'une image 2D, 3D ou de données multispectrales, la présence d'un grand nombre de données régulières du même type est la première à pointer pour le traitement sur les données en parallèle.

3.2.5 Parallélisation d'exécution

Un autre phénomène entre en jeu. On l'appelle *la parallélisation des tâches* ou *la parallélisation d'exécution*. Il ne suffit pas d'avoir une grande quantité de données pour pouvoir calculer en parallèle, il faut également que le calcul à effectuer (les tâches) puisse être exécuté sur plusieurs données en même temps.

Les procédés de traitement dans le domaine de la morphologie mathématique sont souvent de ce type. On identifie assez facilement les parties des algorithmes pouvant être parallélisées dans leur exécution. Il s'agit des prescriptions du genre "*pour tous fait...*", "*pour tous les pixels effectue...*", etc.

Un exemple intéressant de parallélisme au niveau des instructions concerne les architectures VLIW. Nous présentons ici le listing du code du processeur ST200[Bag02], q.v. fig. 3.8. L'exécution se poursuit par blocs délimités par double point-virgule (" ; ;") et les instructions de chacun de ces blocs sont exécutées en même temps. Dans la figure présentée, nous avons encadré trois de ces blocs, chacun ayant la durée d'exécution de 1 cycle d'horloge et chacun exécutant 4 instructions à la fois. La planification de la parallélisation d'exécution est accomplie par le compilateur au moment de compilation. Ainsi, elle est explicitement présente dans le code du programme. L'influence de cette approche sur la performance est évidente, comme son impact sur la structure de l'architecture où la parallélisation ne concerne que les unités exécutives. L'unité de préparation des instructions reste une seule et commune car elle décode une seule large instruction.

3.2.6 Écartement et latence des instructions

Les instructions exécutées par un ordinateur, que ce soit un ordinateur d'une architecture RISC ou CISC, ont un comportement temporel décrit par deux paramètres. Il s'agit de l'*écartement* (angl. pitch) et de la *latence* (angl. latency) des instructions. Ces paramètres sont exprimés en cycles et ils sont d'une importance cruciale sur les architectures avec un pipeline d'exécution.

Le premier paramètre, l'*écartement* d'une instruction, nous indique le temps, mesuré en cycles d'horloge, qui doit passer entre le lancement de l'instruction A et le lancement de l'instruction suivante B si

```
0x8004b3c <main+100>:    add $r17 = $r16, 5 (0x5)
0x8004b40 <main+104>:    add $r11 = $r16, 6 (0x6)
0x8004b44 <main+108>:    add $r15 = $r15, -8 (0xfffffff8);;
0x8004b48 <main+112>:    stb 0 (0x0)[$r21] = $r20
0x8004b4c <main+116>:    add $r18 = $r19, $r18
0x8004b50 <main+120>:    add $r17 = $r19, $r17
0x8004b54 <main+124>:    add $r10 = $r16, 7 (0x7);;
0x8004b58 <main+128>:    stb 0 (0x0)[$r18] = $r20
0x8004b5c <main+132>:    cmpgt $b0 = $r15, $r0
0x8004b60 <main+136>:    add $r11 = $r19, $r11
0x8004b64 <main+140>:    add $r16 = $r16, 8 (0x8);;
0x8004b68 <main+144>:    stb 0 (0x0)[$r17] = $r20
0x8004b6c <main+148>:    add $r10 = $r19, $r10
0x8004b70 <main+152>:    add $r18 = $r19, $r16
0x8004b74 <main+156>:    add $r17 = $r16, 1 (0x1);;
0x8004b78 <main+160>:    stb 0 (0x0)[$r11] = $r20;;
0x8004b7c <main+164>:    br $b0, 0x8004b18 <main+64>
```

FIG. 3.8 : Listing d'un programme pour le processeur ST200. L'exécution se poursuit par des blocs encadrés, ici par 4 instructions à la fois, chacun des blocs est exécuté durant 1 cycle d'horloge.

les opérandes de l'instruction B ne dépendent pas des résultats de l'instruction A. Le temps exact diffère suivant les architectures, il est égal à au moins 1 cycle sur les architectures à lancement simple des instructions, telles que SH-5. Nous pouvons citer les architectures superscalaires comme un exemple non trivial où la définition de l'écartement n'est pas évidente car ces architectures peuvent assurer l'exécution de 2 instructions et plus en même temps.

Le deuxième paramètre, la *latence* d'une instruction, nous indique le temps, mesuré en cycles d'horloge, qui doit passer entre le lancement de l'instruction A et le lancement de l'instruction B si l'instruction B utilise comme opérande un des résultats de l'instruction A. Dans ce cas, la préparation de lancement de l'instruction B est suspendue jusqu'à ce que la latence exigée soit assurée.

C'est le compilateur qui devrait assurer la disparition des temps morts issus des latences des instructions. Cette disparition est réalisée, si réalisable, en changeant l'ordre d'exécution des instructions et en plaçant une instruction à longue latence dans une séquence d'instructions à latence faible ou sans latence.

3.2.7 Instructions spécialisées

Un jeu d'instructions bien choisi incluant des instructions spécialisées peut rendre, s'il est bien utilisé, une architecture très performante.

Nous pouvons mentionner, par exemple, le jeu d'instructions multimédia sur les architectures GPPMM pour le calcul sur les données multiples. Nous pouvons mentionner également le jeu d'instructions mathématiques spécialisés sur les GPU.

Les algorithmes du domaine de la morphologie mathématique travaillent avec l'approche ensembliste de traitement d'images et utilisent donc abondamment les fonctions logiques, fonctions de comparaison et pour le travail en niveaux de gris les fonctions mathématiques max et min. Les constructions du type *if-else* sont également très courantes mais ne sont pas adaptées à certaines architectures car elles sont traduites et exécutées comme les sauts qui peuvent rendre difficile l'exploitation du pipeline et la gestion des mémoires caches. On les transforme facilement en une séquence non conditionnelle en utilisant les instructions de comparaison et de déplacement conditionnel.

Nous allons viser les architectures qui ont ces fonctions incorporées nativement dans leurs jeux d'instruction. Soulignons que le choix même de l'architecture matérielle ayant les instructions désirées ne suffit pas. Un choix de la même importance est celui du compilateur pour cette architecture et surtout sa qualité. Nous avons beau avoir une architecture puissante, c'est le compilateur qui la valorise pour l'utilisation et pour l'exécution des programmes.

3.2.8 Nombre de registres et leur désignation

Lors d'un traitement multimédia qui travaille avec un grand nombre de données originale et de résultats partiels en même temps, le nombre suffisant de registres qui seraient directement utilisables comme

Architecture	Nombre de registres
Intel IA-32 Pentium 4	8 Mif64, 8 Mif128
Intel IA-64 Itanium 2	128 Mi64, 128 Mf82
IBM PowerPC 970FX (G5)	32 Mi64, 32 Mf32, 32 M128
SuperH SH-5	63 Mif64, 1 spécial Mif64

Legende : M* – registre multimédia, *i* – de types de nombres entiers, *f* – de types de nombres en virgule flottante, *64 – registres de 64 bits, *82 bit – registres de 82 bits, *128 – registres de 128 bit

TAB. 3.1 : Le nombre et la désignation des registres multimédia des représentants des architectures grand public

les arguments d'entrée d'une instruction arithmétique ou logique est indispensable. Ce besoin est accentué encore plus lors de traitement multimédia de données par l'approche qui découpe l'image aux macro blocs où nous avons besoin de manipuler un volume important de données vectorielles en même temps. La table 3.1 présente une liste des types et de nombre de registres pour les architectures multimédia les plus représentatives pour les applications grand public.

Notons que l'architecture Intel IA-32 emploi un autre modèle de fonctionnement qui utilise abondamment le travail avec la mémoire cache et où nous avons la possibilité d'utiliser une donnée stockée dans la mémoire directement comme argument d'une instruction arithmétique. Donc, cette architecture compense le nombre relativement petit (si on le compare avec l'architecture Intel IA-64) de registres par un accès rapide aux données stockées dans la mémoire cache L1 qui est, généralement, intégrée sur la puce et cadencée, généralement, à la même fréquence que celle du processeur.

3.2.9 Approche à l'implémentation des algorithmes

Lors du développement d'une application qui devrait inclure les instructions spécialisées et qui vise la performance maximale sur une architecture donnée, c'est-à-dire qui vise la rapidité maximale d'exécution d'un programme sur cette architecture, au moins trois approches sont possibles :

- La première approche consiste en l'utilisation des fonctions déjà prêtes et écrites spécifiquement pour l'architecture cible par des professionnels. Nous pouvons citer les bibliothèques des fonctions telles qu'IPP[Int06b] d'Intel, une bibliothèque des primitives optimisées. Le gain de temps est évident, on épargne le développement du code spécialisé. En revanche, ces bibliothèques ne fournissent pas, en général, un ensemble complet des fonctions dont on aurait besoin dans l'application. Dans les applications plus complexes et plus compliquées, on ne peut pas se contenter seulement de ces bibliothèques et on est obligé de créer nos propres fonctions spécialisées. Nous pouvons citer l'exemple de l'outil de recherche *MorphoMedia*[Bra05], développé au Centre de Morphologie Mathématique à Fontainebleau au cours du projet de cette thèse.
- La deuxième approche consiste à faire confiance au compilateur et en sa qualité en espérant qu'il reconnaîtra automatiquement les constructions complexes qui peuvent être présentes dans les programmes écrits manuellement par les programmeurs. C'est l'approche a priori la plus rapide en temps de programmation de nos propres fonctions mais elle ne garantit pas nécessairement le résultat attendu. Pour éviter les problèmes liés au temps d'exécution à l'échéance d'un projet de développement informatique, il faut adopter dès son début un style de programmation propre aux exigences et à la compréhension du compilateur. Cela implique un petit surplus de temps utilisé pour comprendre les détails de fonctionnement du compilateur. Il s'agit le plus souvent des options de compilation non standards et des directives du compilateurs incorporées directement dans le code. Par exemple, dans le langage C++ il s'agit des directives *#pragma*.
- La troisième approche consiste en programmation de très bas niveau et en une utilisation directe des instructions d'assembleur de la machine. C'est l'approche la plus coûteuse en temps de développement mais elle est la plus proche de l'architecture et elle peut exploiter au maximum ses particularités. En contraste avec les approches précédentes, elle exige du personnel connaissant les détails et les points spéciaux de l'architecture.

La portabilité d'un programme qui utilise les instructions spécialisées à une autre architecture est un autre sujet à discuter. Le code écrit pour une architecture particulière qui, de plus, s'en sert couramment, est difficile à porter à une autre architecture. Le grand avantage des instructions multimédia des architectures GPPMM est que pour quasiment tous les représentants existants des GPPMM nous trouvons, dans leurs jeux d'instructions, les instructions multimédia qui assurent des fonctionnalités identiques ou très similaires.

De ce point de vue, la solution de portage peut être réalisée à l'aide de la couche d'abstraction du matériel, HAL, une couche intermédiaire au niveau du logiciel qui encapsule les différences des codes assembleurs de différentes architectures et n'exporte qu'un interface unifié et généralisé. Tel est le cas pour l'outil de recherche MorphoMedia [Bra05]. Les GPU, eux aussi, sont dotés de la couche HAL présentée par le pipeline graphique abstrait au niveau du logiciel.

3.3 Consommation d'énergie

Un paramètre important pour le choix d'une architecture et qui est étroitement lié à la performance que l'on vient de discuter dans 3.2, est la consommation de l'énergie. Pour certaines applications, la consommation ne joue pas un rôle important et le seul facteur limitant pourrait être le coût de l'électricité. Pour d'autres, la consommation d'énergie est le facteur majeur du choix du matériel, surtout pour les applications portables et nomadiques où l'équipement matériel n'est pas toujours connecté à une alimentation fixe et où ont doit gérer la consommation, d'une façon ou l'autre. Cela est assuré souvent par l'utilisation des régimes économiques qui changent la fréquence du processeur selon les besoins actuels de performance ou qui arrêtent partiellement ou complètement le fonctionnement en cas d'inactivité. Dans ces cas précis, le taux consommation/performance est celui qui oriente le choix d'un processeur plutôt que d'un autre.

La table 3.2 présente une liste non exhaustive des processeurs issus des architectures que nous ciblons dans cette thèse. Le premier groupe présente la consommation de processeurs GPP/GPPMM. Le deuxième groupe, en contraste avec le premier, présente les GPP/GPPMM mobiles ou à basse consommation. Le troisième groupe présente la consommation des GPU. Le tableau finit par présenter la consommation des consoles de jeux, c'est-à-dire d'autres architectures spécialisées qui peuvent être visées par les algorithmes décrits dans cette thèse.

Ce qui nous intéresse quant aux processeurs dans ce tableau, c'est la consommation d'énergie. Pour l'information, nous y présentons également les valeurs de MIPS et de FLOPS que nous avons pu collecter pour que le lecteur se fasse une idée des performances liées à la consommation. Tout en sachant que ces descripteurs ne sont pas les meilleurs que l'on puisse trouver pour évaluer la performance mais ils nous permettent de comparer, au moins au niveau de l'ordre, les performances d'architectures aussi différentes que, par exemple, SH-5 et AMD Athlon™64 FX-60 avec deux cœurs.

Le premier regard sur les données dans le tableau 3.2 nous révèle que la consommation des GPPMM, des GPU et des consoles de jeux est de même ordre. Comme les vainqueurs sortent de cette comparaison les consoles de jeux car si on voulait obtenir l'équivalent d'une telle solution à partir des GPP et GPU, nous devrions additionner la consommation d'un GPP choisi avec la consommation d'un GPU choisi ce qui donnerait à l'estime une somme deux fois supérieure à celle des consoles de jeux. Dans une telle logique, le ratio performance-consommation est plus favorable aux consoles de jeux.

Les solutions à consommation basse ou réduite se font remarquer par la consommation inférieure de quelques ordres par rapport aux solutions décrites précédemment. Ce qui n'est pas surprenant car il s'agit de leur désignation. En même temps, ils n'offrent pas les mêmes performances. La différence est importante mais le ratio MIPS par Watt est plus élevé pour les solutions à basse consommation.

Parmi les architectures de ce type et tout-à-fait intéressantes qui combinent une haute performance et une basse consommation d'énergie nous trouvons les architectures VLIW. Ces architectures sont représentées par les processeurs tels que ST200 [Bag02] basés sur la plateforme nommée Lx [FBF00] de STMicroelectronics et par les processeurs Transmeta Crusoe [Tra05a] et Efficeon [Tra05b]. Le dernier, Transmeta

GPP/GPPMM

Processeur(s)	Année	Fréquence, MHz Processeur (/ Mémoire)	Consommation, W	MIPS	MFLOPS
Intel Pentium XE 955 [Int06c] (dual cœur, HT)	2006	3460 / 1066	156 [Gav05a]	—	—
AMD Athlon 64 FX-60 [Gav06] (dual cœur)	2006	2600	110	22150 [Wik06h]	—
Intel Pentium 4 6xx [Gav05b]	2005	3000 à 3800	120 – 140	—	—
Intel Itanium 2 [NSG05b]	2005	1800	100	—	—
AMD Athlon 64 [Wik05a]	2005	2800	—	8400	—
Intel Pentium 4 EE [Wik06h]	2003	3200	—	—	9726

GPP/GPPMM mobiles, à basse consommation

Processeur(s)	Année	Fréquence, MHz Processeur (/ Mémoire)	Consommation, W	MIPS	MFLOPS
Transmeta Efficeon TM8800 [Tra05c] (VLIW)	2005	1700	3W@1GHz [Kra04]	—	—
ARM11 MPCore [GS05, ARM06] multiprocesseur 4x core, SIMD	2005	400-550	<0.25	2600 (DMIPS)	—
SuperH SH-5 [Nor01, Sup02]	2002	133-167-400	< 0,4	240-300-714	—
Mobile Pentium III [Nor01]	2001	750	20	875	—
PowerPC 750 [Nor01, FS02]	2001	200-400	5,7-5,8	488-733	—

GPU

Processeur(s)	Année	Fréquence, MHz Processeur (/ Mémoire)	Consommation, W	MIPS	MFLOPS
NVidia 7800 GTX Extreme [Kre05]	2005	430	180-343	—	—
NVidia 7800 GTX [TP05]	2005	430	100-110 [Rey06]	—	165000
2x NVidia 7800 GT Dual (4 GPU) [Kre05]	2005	430	213 / 317 (2D / 3D)	—	—
ATI Radeon X850	2004(5)	540 / 590 GDDR3	90 [ATI06]	—	—

Consoles de jeux

Processeur(s)	Année	Fréquence, MHz Processeur (/ Mémoire)	Consommation, W	MIPS	MFLOPS
Sony PlayStation 3 [Wik06i]	2005(6)	3200 (8xCPU), et 550 (GPU) / 3200 XDR DRAM, 700 GDDR3	—	—	218000
Microsoft Xbox 360 [Wik06l]	2005	3200 (3xCPU), 500 (GPU) / 1600 L2, 700 GDDR3	160 [Gre05]	—	115000

TAB. 3.2 : Consommation d'énergie des GPP/GPPMM, des GPU et des consoles de jeux

Efficeon, devrait consommer 3W à 1 GHz[Kra04] et implémente les instructions multimédia[Tra05c], compatibles avec MMX, SSE-SSE3 d'Intel ce qui entre intégralement en cohérence avec le sujet de cette thèse et ouvre un autre champ d'application pour nos algorithmes.

3.4 Modèle stream du calcul et les architectures associées

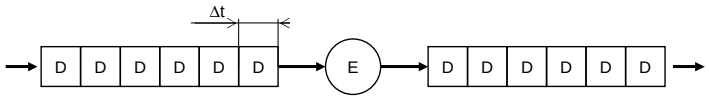
3.4.1 Calcul sur les streams

Le calcul qui considère les données comme si elles étaient alignées dans un flux et arrivaient continuellement dans les unités du calcul est propre aux architectures stream et dans la taxonomie de Duncan correspond, sans surprise, au type *à flux de données*, cf. 3.1.2.4.

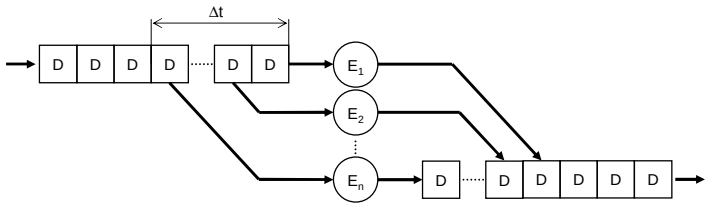
Toutes les données qui passent par une unité exécutive sont traitées de la même façon par un et même programme. Il s'agit d'une caractéristique forte et très importante car elle nous permettra de facilement paralléliser l'exécution à l'intérieur d'une unité du calcul par la multiplication de nos moyens matériels. Ainsi, plusieurs blocs d'exécution à l'intérieur d'une unité du calcul vont pouvoir travailler concurremment en exécutant le même programme, en se partageant les données du flux d'entrée et en assemblant les résultats dans un flux de sortie.

Si le traitement d'une donnée est indépendant des autres données, la parallélisation se révèle simple. Cette situation est démontrée sur la fig. 3.9. Sa partie 3.9(a) illustre le traitement d'un stream de données où ce n'est qu'une seule donnée qui est traitée dans un temps Δtet indépendamment des autres données. La parallélisation de ce traitement est présentée sur la fig. 3.9(b) ; dans le même intervalle Δt, n données du flux d'entrée sont distribuées, traitées dans les unités exécutives et assemblées à nouveau dans le flux de sortie.

Notons que dans le cas où le traitement d'une donnée de sortie dépend de plusieurs données d'entrée (e.g. la somme des éléments), la stratégie de parallélisation ne sera pas aussi intuitive et va fortement dépendre du caractère de l'opération à effectuer.



(a) exécution en stream avec 1 unité exécutive



(b) exécution en stream et en parallèle avec *n* unités exécutives

FIG. 3.9 : Calcul sur un stream de données, D - donnée, E_i - unité exécutive i

D'une façon générale, nous ne parlons pas d'une opération ou d'un programme mais plutôt d'un *kernel*. Kernel est un terme qui englobe toutes les activités qui peuvent être effectuées sur un stream. Regardons maintenant quels sont les types d'opérations les plus courantes du paradigme stream que les kernels implémentent. Il s'agit de l'*application* (angl. *map*), de la *réduction* (angl. *reduce*), et du *filtrage* (ang. *filtering*). Davantage de types de kernels opérant sur les stream peut être consulté dans la littérature[OLG+05]. Remarquons que ces opérations ont leurs correspondants directs dans les fonctions de divers langages de programmation qui supportent, d'une façon ou de l'autre, le travail avec les listes. Notamment il s'agit de Haskell, LISP, Python.

3.4.1.1 Kernel d'application

L'application est une opération élémentaire et en même temps la plus simple qui peut être effectuée par un kernel. Étant donné un stream S et une fonction f, le kernel K_M implémentant l'opération *application* va appliquer la fonction f à tous les éléments du stream S. La fig. 3.10 illustre cette situation.

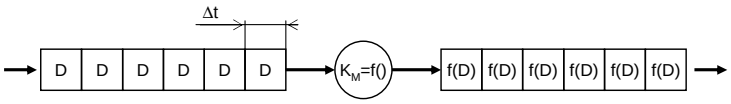


FIG. 3.10 : Exemple d'un kernel d'application. La fonction f est appliquée à tous les éléments du stream. D - donnée, K_M - kernel d'application, f - fonction à appliquer

3.4.1.2 Kernel de réduction

Le kernel de réduction effectue une opération qui à partir d'un large stream d'entrée dont les éléments sont constitués de m données élémentaires calcule un stream plus étroit de n données élémentaires où $m > n \geq 1$. Nous pouvons citer comme exemple toutes les fonctions ayant plusieurs valeurs à l'entrée et une valeur à la sortie, par exemple la somme, somme des valeurs absolues, maximum, minimum, opération sur le voisinage, opération binaire avec un masque, etc. La fig. 3.11 illustre cette situation.

Un exemple trivial d'un kernel de réduction est la somme des valeurs du stream tout entier. Son résultat est un flux constitué d'un seul élément qui ne contient qu'une seule donnée - la somme.

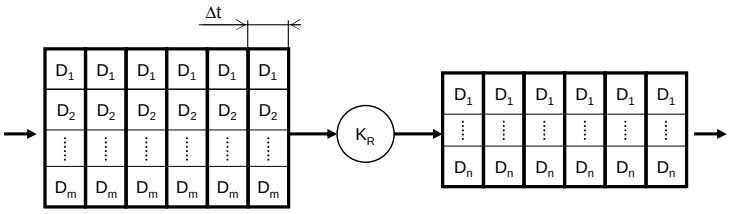


FIG. 3.11 : Le kernel de réduction crée un stream plus étroit à partir d'un stream plus large. D - donnée, K_R - kernel de réduction

3.4.1.3 Kernel de filtrage

Le kernel de filtrage effectue un filtrage sur le stream en laissant passer les éléments qui remplissent une condition et en éliminant les autres. Nous pouvons citer comme exemple le filtrage pour éliminer les valeurs négatives d'un stream, q.v. fig. 3.12.

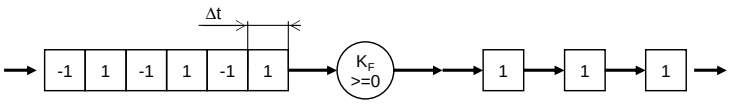


FIG. 3.12 : Exemple de fonctionnement d'un kernel de filtrage. À la sortie, les valeurs négatives sont éliminées du stream. D - donnée, K_F - kernel de filtrage

3.4.2 Architectures stream

Les *architectures stream* sont les architectures qui ont été directement conçues pour le travail avec les flux de données. Les exemples les plus éloquents des systèmes travaillant sur des données stream sont sûrement les composantes de réseau. Dans leurs cas, les données arrivent dans un stream en tant que paquets, ordonnées dans le temps. Ces paquets sont traités par le programme de la composante et ressortent comme des paquets transformés, également dans un stream.

Les architectures de ce type sont particulièrement intéressantes pour le domaine du traitement d'images et des signaux. La télévision et le traitement de la vidéo sont de très bons exemples des applications où le flux de données est traité par les architectures stream. Nous pouvons mentionner aussi les capteurs CCD ou CMOS présents dans les équipement digitaux tels que les caméras ou les appareils photographiques qui ont à leur sortie un flux linéaire des données. Ces données, une fois stockées comme des images 2D dans la mémoire, ne présentent plus la nature d'un flux vidéo. Cette nature revient lors d'un traitement de ces données sur les processeurs séquentiels où un procédé très fréquemment utilisé est constitué de la transformation de ces données en un flux suivie par l'application d'un algorithme sur ce flux. Notons que ce flux peut être constitué des pixels mais également des unités plus grandes comme des lignes entières ou les macro blocs. Comme exemple nous pouvons citer une architecture reconfigurable *Cheops* [BW95], conçue pour le traitement des données vidéo dans les caméras et les télévisions HDTV.

Le phénomène de traitement sur les flux est présent également dans les traitements morphologiques, nous pouvons citer comme exemples une architecture électronique dédiée aux algorithmes de segmentation [Lem96] ou une étude des architectures spécifiques pour la morphologie [Sas02] ou même une réalisation antécédente (1983/84) d'un logiciel de morphologie mathématique pour le calculateur PROPAL 2 [BM83, Beu84], spécialisé pour le traitement en parallèle avec les capacités SIMD.

Plus récemment, l'équipe commune des laboratoires des université de Stanford et de Rice a présenté plusieurs publications portant sur l'architecture d'un processeur stream nommé Imagine[1] qu'ils ont développée. Ce processeur était doté de 8 clusteurs de traitement général et avec 6 ALUs par clusteur : il possédait 48 ALUs au total. Ce processeur était réalisé sur une seule puce pour pouvoir exploiter au maximum la localité des données stockées dans les registres temporaires présents sur la puce.

Bien que ces architectures soient spécifiques et même très intéressantes pour les applications de nos algorithmes et bien que nos algorithmes s'adressent aussi à elles, ces architectures ne constitueront pas notre intérêt principal et nous ne les mentionnons ici que pour donner une vue globale et complète sur la problématique. Car, comme nous l'avons souligné dans le chapitre *Motivation*, page 19, nous nous intéressons en particulier aux architectures plus générales grand public avec les fonctionnalités multimédia et aux processeurs graphiques.

3.4.3 Architecture de von Neumann et ses successeurs utilisés pour le calcul stream

3.4.3.1 Architecture de von Neumann

Les machines du calcul que nous connaissons aujourd'hui en tant que GPP sont basées sur l'architecture de von Neumann qui se caractérise par un bloc d'unité centrale (CPU) qui assure le calcul, par un bloc de mémoire commune pour les données et pour les instructions et par un bus qui relie ces deux blocs ensemble. La figure 3.13 présente la structure de l'architecture du type von Neumann avec la mémoire cache présente dans le CPU.

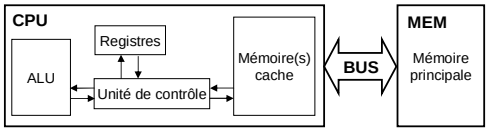


FIG. 3.13 : Schéma de l'architecture von Neumann avec une mémoire cache incorporée dans l'unité centrale, CPU - bloc d'unité centrale, MEM - bloc de la mémoire, BUS - bus assurant la liaison de l'unité centrale à la mémoire

Il faut mentionner intensément que l'architecture de von Neumann n'est pas une architecture parallèle. Elle était conçue pour l'exécution d'un code séquentiel et dans sa version d'origine, elle n'est pas du tout adaptée à un calcul sur les streams. Notons que le code parallèle est exécuté en séquence sur

[1] Imagine Stream Processor [ORK+02, KDK+01, KDR+02, KRD+03], implémenté physiquement en silicium sur la puce dont la surface était 2,6 cm^2

cette architecture et nous n'exploitons pas du tout les avantages d'un possible parallélisme. Le goulot d'étranglement principal et en même temps un grand désavantage de cette architecture est la manière d'accéder aux données dans la mémoire où celles-ci sont obligées de circuler dans les deux sens entre la mémoire et l'unité du calcul. La mémoire cache présente sur la puce dans le bloc CPU est déjà une technique utilisée pour diminuer l'impact de ce goulot d'étranglement.

En revanche, les architectures à flux de données n'utilisent pas deux concepts fondamentaux de l'architecture de von Neumann – le *compteur des instructions* (angl. Program counter), qui est propre au traitement séquentiel, et le stockage global des données dans un système de mémoire composé d'une hiérarchie plus ou moins complexe des mémoires.

Les solutions qui seraient convenables à la fois pour le calcul en séquence et pour le calcul sur les streams et qui essayaient de combiner l'architecture de von Neumann et l'architecture de flux de données ont été également étudiées, surtout dans les années 1980. Diverses réalisations qui travaillaient avec différents niveaux de granularité ont été présentées ; comme exemple nous pouvons citer l'architecture hybride décrite dans la thèse de Iannucci [Ian88] en 1988.

De nos jours, l'approche percevant l'exécution du code dans le temps comme des *fils* d'exécution (angl. thread) a largement changé le point de vue sur la structure originale de l'architecture de von Neumann et a orienté l'évolution des architectures GPP basées sur cette dernière vers les architectures adaptées au travail avec les flux de données – architectures *multithread*. Pour plus d'information sur le rapprochement entre des architectures à flux et des architectures *multithread*, nous invitons le lecteur se référer à un article [SRU01] qui étudie ce point précis très en détail. Pour plus de détails généraux sur les diverses architectures des ordinateurs et sur leurs conceptions, nous adressons le lecteur à une des livres de William Stallings, e.g. la septième[1] édition [Sta06].

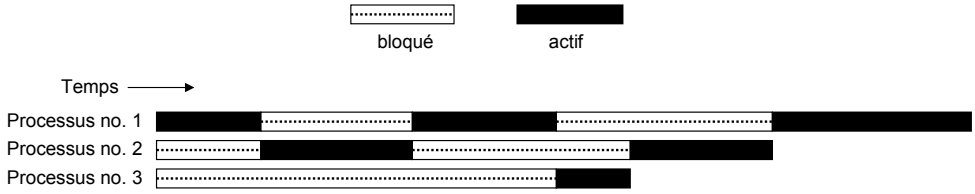
3.4.3.2 Fils d'exécution et multitâche

Percevoir l'exécution sur un ordinateur à travers des *processus* distincts (on parle aussi des *tâches*) est la pratique couramment utilisée dans les systèmes logiciels qui doivent assurer l'exécution des programmes ou routines d'une façon concurrente. Ils doivent souvent gérer la priorité, notamment dans les cas où il est important de livrer les résultats d'une tâche soit le plus tôt possible, soit dans un temps prédéfini, soit plus tôt que le résultat d'une autre tâche. L'exemple le plus marquant de travail avec les processus que nous appelons également travail *multitâche* sont les systèmes d'exploitation multitâches dans les ordinateurs. Mais nous pouvons trouver le même style de travail aussi dans les systèmes informatiques pour l'automatisation industrielle où les tâches sont également concurrentes et de plus, la demande d'obtention des résultats dans un temps prédéfini est cruciale et dicté par la nature de la technologie à contrôler.

Pour exploiter le parallélisme à une échelle plus petite à l'intérieur d'un processus, les *fils d'exécution* ou autrement dits les *processus légers* [Wik06j] (angl. threads) sont utilisés. La différence entre les fils d'exécution et le multitâche classique réside dans le fait que les processus classiques sont typiquement indépendants les uns des autres et communiquent en utilisant une interface fournie par le système. En revanche, les fils d'exécution peuvent partager des information sur l'état du processus auquel ils appartiennent, les espaces de la mémoire ainsi que d'autres ressources. L'exemple trivial d'exécution en fil est celui d'un processus constitué d'un seul fil d'exécution, d'un *single thread*. Dans les autres cas, nous parlons d'un processus à plusieurs fils d'exécution, d'un processus *multithread* (MT).

L'architecture matérielle qui effectue l'exécution peut être, et pour les GPP fut longtemps, l'architecture de von Neumann. Puisqu'il ne s'agit pas de l'architecture parallèle, l'aspect concurrent d'exécution disparaît, le temps du processeur est distribué par portions entre les fils et l'exécution se poursuit d'une façon séquentielle. Mais à l'échelle du temps moins précise, l'exécution des fils est perçue comme concurrente. Dans ce cas nous parlons du *multithreading virtuel* (VMT).

[1] Septième édition est la plus récente en 2006



(a) Multiprocessing sur un processeur

(b) Multiprocessing sur plusieurs processeurs, les processus sont exécutés concurremmet

FIG. 3.14 : Exécution de plusieurs processus sur les architectures différentes (selon Stallings [Sta06])

Si les fils peuvent être exécutés en même temps, nous parlons du *multithreading simultané* (SMT). Sur les architectures où l'aspect parallèle d'exécution est présent et qui peuvent lancer plusieurs instructions de plusieurs fils d'exécution, nous parlons du *Chip-level MultiThreading* (CMT). Si plusieurs processeurs sont intégrés dans une seule puce, nous parlons de *multiprocessing au niveau de la puce* (CMP) ou de *plusieurs cœurs* (angl. multicore).

La sauvegarde de l'état d'un fil d'exécution suivie par la restauration de l'état d'un autre fil qui poursuivra l'occupation des ressources communes lors de l'exécution concourante est nommée le *changement du contexte*. Il peut être moins coûteux, mesuré par la durée de ce changement, sur les architectures qui partagent certaines ressources matérielles comme la mémoire cache, par exemple. C'est souvent le cas pour le multithreading simultané dans les applications multimédia.

3.4.3.3 Architectures à fils d'exécution

Les architectures matérielles se sont adaptées à cette façon de travailler et ont procédé aux améliorations en éliminant progressivement les goulots d'étranglement qui freinaient les applications logicielles de ce type.

Dans un premier temps, l'effort s'est concentré sur les unités de préparation d'exécution des instructions. Ces unités travaillent en pipeline qui peut être relativement long[1]. Durant le travail avec les threads, le changement du contexte provoque le vidage de ce pipeline et un nouveau remplissage par l'état restauré appartenant au nouveau thread. Ce qui cause, en effet, la perte du temps. Cette perte est directement proportionnée, à travers les cycles d'horloges nécessaires pour remplir le pipeline à nouveau, à la longueur de ce pipeline et à la fréquence de changements des contextes. Ce qui défavorise fortement les applications travaillant fréquemment avec les threads. Pour diminuer cette perte, la solution se trouve dans la multiplication du nombre de ces pipelines. Les processeurs qui ont adopté cette solution implémentent, en effet, le multithreading simultané. La technologie de ce type développée par Intel est nommée Hyper-Threading Technology [MBH+02].

La fig. 3.15 présente le schéma de la structure d'une telle architecture. Remarquons que ces technologies, que nous classons sous CMT, ne sont pas destinées uniquement aux processus légers mais exploitent aussi bien le parallélisme des processus distincts lors d'un travail multitâche.

Les solutions récentes utilisent cette idée et ont aussi étendu la multiplication matérielle aussi aux unités centrales entières. C'est ainsi que nous avons aujourd'hui à notre disposition les architectures à

[1] Tejas, le prototype du successeur du Intel Pentium 4, dont le développement a été abandonné, possédait un pipeline de 31 phases [FH05]

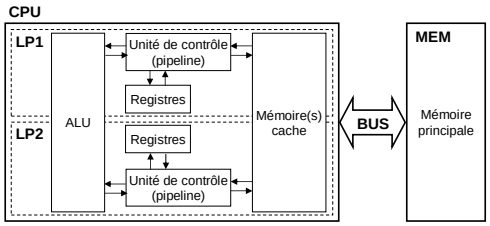


FIG. 3.15 : Architecture hyper-threading, CPU - bloc d'unité centrale, MEM - bloc de la mémoire, BUS - bus liant l'unité centrale avec la mémoire, LP_* - processeur logique

plusieurs cœurs qui implémentent plusieurs unités centrales à l'intérieur d'une seule puce. Si les cœurs eux-mêmes implémentent le multithreading simultané, nous pouvons parler de la combinaison du multiprocessing, CMP, et du multithreading, CMT, au niveau de la puce. La fig. 3.16 illustre cette situation sur un processeur avec deux cœurs où chaque cœur implémente aussi le multithreading simultané. Nous pouvons citer comme représentant les processeurs avec deux cœurs d'Intel[Int06d] ou d'AMD[AMD06] et le processeur à basse consommation avec quatre cœurs d'ARM[ARM06].

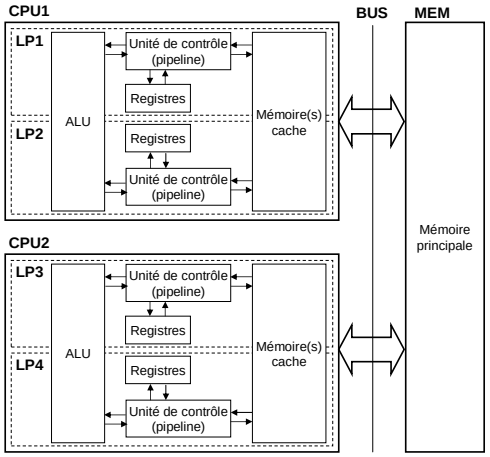


FIG. 3.16 : Architecture hyper-threading avec deux cœurs, CPU_* - bloc d'unité centrale, MEM - bloc de la mémoire, BUS - bus liant l'unité centrale avec la mémoire, LP_* - processeur logique

Ces solutions présentent l'aspect parallèle et nous pouvons les classer parmi les architectures MIMD avec la mémoire partagée. La philosophie de construction a changé, on est en train de travailler en parallèle mais nous pouvons toujours distinguer les bases de l'architecture de von Neumann – un bloc de mémoire d'un côté et un bloc d'unité centrale, même complexe et avec deux cœurs, de l'autre.

3.4.4 Calcul stream sur les architectures SWAR à plusieurs fils

Comment exploiter alors le parallélisme présent dans les architectures succédant à celle de von Neumann pour le travail qui nous intéresse ? Puisqu'il s'agit des architectures parallèles qui exploitent le parallélisme de plusieurs flux d'instructions, nous devrions modifier notre regard classique sur l'exécution en stream, comme présenté dans 3.4.1, page 44, sans pour autant modifier notre philosophie de travail.

En effet, nous devons exprimer le calcul d'un kernel d'exécution sur un seul flux de données en utilisant plusieurs fils qui vont tous implémenter la même fonctionnalité du kernel et travailler sur les parties du flux principal.

On procède à la division du problème en parties plus petites, suivie par l'exécution en parallèle sur une architecture parallèle, et terminons par la collecte des résultats. Ce procédé correspond au paradigme de la programmation parallèle nommé *Divide and Conquer*, propre aux architectures MIMD, où une et

même routine est exécutée sur plusieurs processeurs, physiques ou logiques, en traitant plusieurs données en même temps. La fig. 3.17 illustre cette situation.

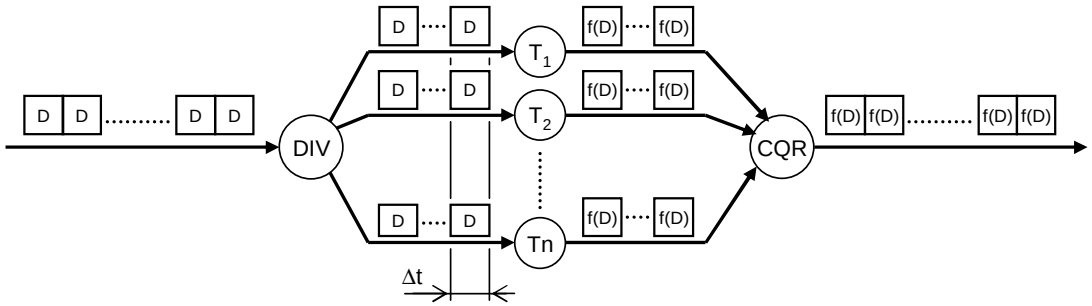


FIG. 3.17 : Calcul sur un stream en utilisant les fils d'exécution et l'approche *Divide ans Conquer*, T_* - thread, DIV - phase de division du stream, CQR - phase conquer, collecte des résultats, D - donnée, f - fonction du kernel

La différence entre le traitement du stream en parallèle comme on l'a présenté dans la section 3.4.1 et sur la fig. 3.9(b), page 44, qui correspond au paradigme nommé *processor farm* ; et le traitement du stream en fils d'exécution qui correspond au paradigme *Divide and Conquer* réside dans le fait que pour le dernier, les unités parallèles de traitement des threads ne peuvent pas être englobées dans un macro bloc et être perçues comme une seule macro unité traitant le stream. En effet, nous divisons physiquement le stream principal en autant de sous-streams qu'il y a de fils d'exécution. Ainsi, le procédé *Divide and Conquer* est bien perceptible.

Les architectures contemporaines GPPMM, cf. 1.2 dans le chapitre Motivation, page 20, utilisent les fonctionnalités SIMD incorporées dans les instructions et implémentent le parallélisme de données au niveau des registres. Nous parlons ainsi du traitement SWAR[Nor01], autrement dit traitement SIMD à l'intérieur d'un registre (angl. *SIMD Within A Register*), qui correspond au *Subword Parallelism* dans le domaine de traitement des signaux. Les GPPMM sont également dotés, outre ces fonctionnalités multimédia, des fonctionnalités matérielles pour l'exécution à plusieurs fils. Elle nous ouvrent ainsi un autre champ d'applications. En exploitant le parallélisme présent dans les traitements d'images par la morphologie mathématique, nous pouvons facilement augmenter la performance finale de nos algorithmes sur ces architectures.

L'utilisation des capacités MT dans les applications du traitement d'images devient progressivement une technique courante[CHD+02] mais son implémentation appartient au domaine de la programmation spécialisée. Nous distinguons deux approches dans le développement de ces applications :

- *développement manuel* qui consiste en programmation d'une application multithread à l'aide des méthodes explicites telles que l'utilisation des bibliothèques pour la création, gestion et synchronisation des threads (Win32 ou POSIX). La restructuration complète du code est exigée pour une telle implémentation.
- *développement automatique ou semi-automatique* à l'aide d'un compilateur. Dans le cas automatique c'est le compilateur lui-même qui reconnaît les structures parallèles et les traduit en threads. Dans le cas semi-automatique c'est le programmeur qui marque, à l'aide des commandes du préprocesseur, les parties du code à paralléliser qui sont ensuite transformées par le compilateur. Il y incorpore ses propres fonctions pour l'exécution des régions en parallèle. Tel est le cas de l'API OpenMP[Ope06] qui est intégré, parmi d'autres, dans les compilateurs Intel[Bre05, TBG+02] du C++ et du Fortran.

3.4.5 Pipeline graphique et les GPUs

3.4.5.1 Terminologie de la synthèse des images

Même si le traitement morphologique des images appartient au domaine de l'analyse d'images, notre explication du pipeline graphique va commencer avec leur synthèse. Car c'est à elle que nous devons la notion du *pipeline graphique*.

Les processeurs graphiques sont les processeurs destinés à la synthèse des images. Ils ont une architecture spécifique, complètement subordonnée au traitement des pixels et destinée à la visualisation à partir d'un modèle de synthèse. C'est pourquoi nous parlons d'un pipeline graphique, d'un pipeline qui est à la fois une abstraction de fonctionnement, implémentée soit dans un logiciel comme un programme soit comme une architecture matérielle où la notion du pipeline prend son véritable sens de chaîne de traitement.

Ainsi, la terminologie de la synthèse des images est utilisée aussi dans la description des algorithmes travaillant avec le pipeline graphique pour désigner les divers blocs fonctionnels de cette dernière et cela même dans les algorithmes qui ne sont pas relatifs à la synthèse des images. Nous allons également suivre cette logique et utiliser cette terminologie dans les algorithmes de la morphologie mathématique pour les GPU où nous ne travaillons pas avec la synthèse des images et nous ne nous concentrons pas seulement sur leur traitement ; nous devrions parler plutôt du traitement des données vectorielles ou matricielles ou du traitement des graphes.

La synthèse des images 2D est un processus qui, à partir d'un modèle mathématique, crée une image 2D destinée principalement à être rendue sur un écran. Le modèle mathématique peut varier, nous pouvons travailler avec les modèles de surface 3D décrits par la géométrie en espace 3D ou avec les modèles de volume 3D qui utilisent l'interprétation d'un volume par des données discrétisées où chaque élément du volume, *voxel*, détient une information. Pour le rendu 3D, il s'agit souvent d'un coefficient d'absorption.

3.4.5.2 Structure du pipeline graphique

Regardons le schéma[1] du pipeline graphique programmable présenté sur la fig. 3.18. Ce schéma présente les blocs principaux du pipeline qui correspondent dans les GPU aux diverses unités de traitement, configurables ou programmables. Ce schéma montre également la façon dont les GPU sont connectés aux GPP et à l'application graphique dans ces derniers à travers de l'environnement run-time d'une interface de programmation d'application (API).

Dans cette configuration, les processeurs graphiques sont les accélérateurs de traitement et sont subordonnés aux processeurs généraux. Ainsi, nous nous retrouvons avec un système de calcul distribué qui ne compte qu'une unité dirigeante et une unité exécutante. Notons qu'à présent, la notion du calcul distribué s'est encore accentuée car nous comptons parmi les réalisations matérielles récentes également les systèmes avec deux ou même quatre GPU [Kre05] : citons les technologie NVidia SLI et ATI CrossFire.

C'est le GPP qui gère complètement le traitement sur les GPU à travers des commandes de configuration, des commandes d'exécution ou des commandes initiant un transfert de données vers ou à partir des GPU. Ces commandes sont passées à travers l'une des interfaces API qui n'expose au programmeur que des fonctions de haut niveau. C'est le pilote de la carte graphique dans les GPP qui reçoit ces commandes, qui les interprète et qui se charge de leur envoi dans les GPU pour les exécuter.

Le traitement sur les GPU est différent de celui que l'on connaît du domaine des GPP. Dans les GPU, les commandes ont la forme des prescriptions à dessiner et c'est, en effet, exactement la manière dont on les programme. L'application envoie la description d'une forme, il pourrait s'agir de points, triangles, rectangles, polygones, courbes ou surfaces. Celles qui ne sont pas reconnues par le matériel sont transformées par les couches intermédiaires de logiciel en formes plus simples. On utilise souvent les lignes ou triangles soit pour diviser une forme complexe en formes basiques (comme un polygone

[1] Schéma synthétisé à partir de plusieurs sources [FK03, FK03, WNDS99]

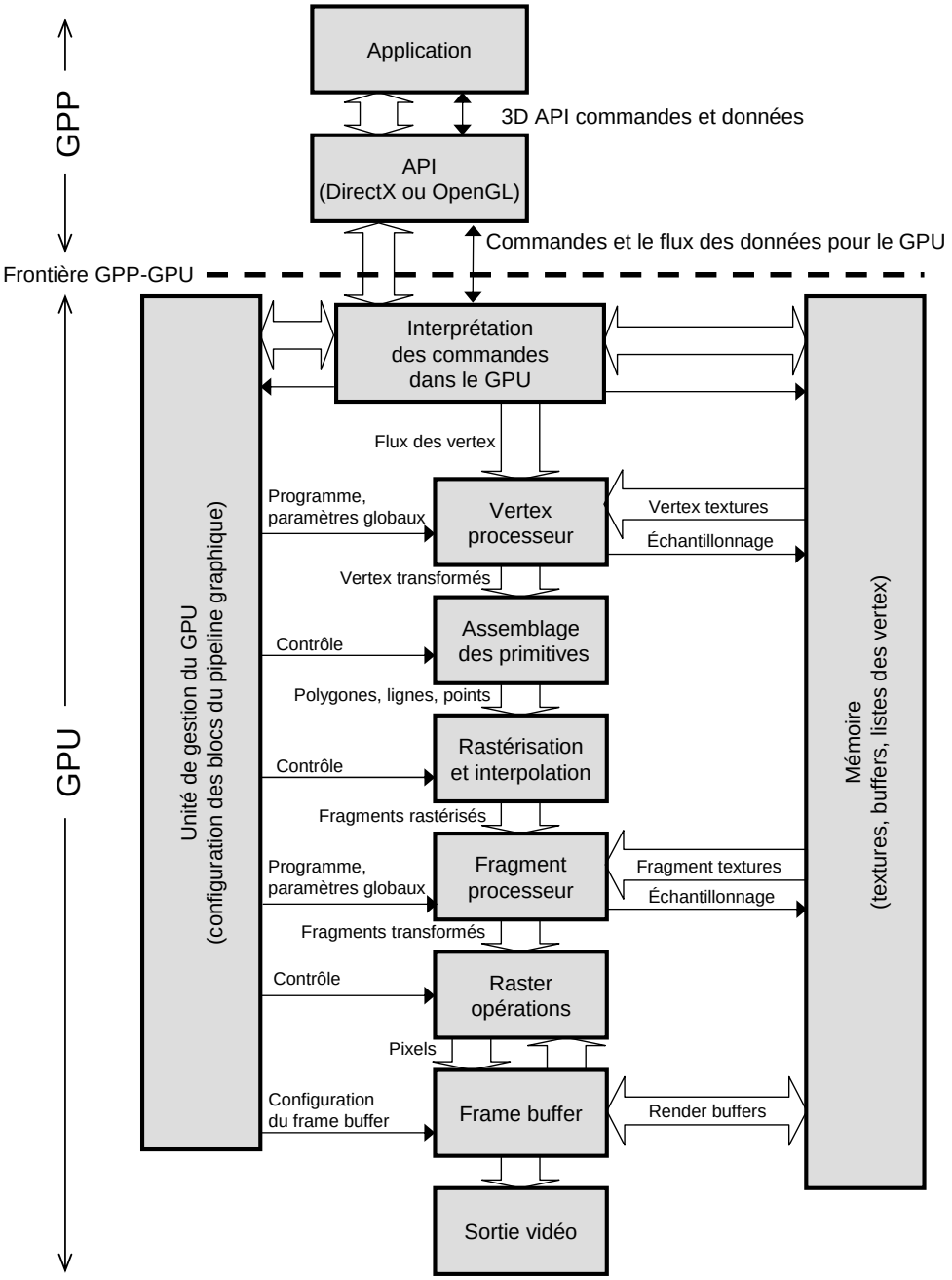


FIG. 3.18 : Schéma des blocs du pipeline graphique

peut être divisé en triangles), soit pour approximer une forme complexe en formes plus simples (comme une courbe spline peut être approximée par plusieurs lignes droites).

Les formes de base sont décrites par leur type (point, ligne, etc.) et par un ensemble de points de l'espace 3D que nous appelons les *vertex*. Un vertex est une structure complexe et elle peut contenir plus d'informations que seulement les coordonnées dans l'espace ; notamment l'information sur la couleur, sur les index pointant dans les textures où même davantage d'informations spécifiques au calcul effectué. Une telle commande graphique est passée, utilisant un des interfaces API, au processeur graphique.

3.4.5.3 Fonctionnement du pipeline graphique

Le fonctionnement de base d'un pipeline graphique programmable est le suivant. Toutes les données sont traitées en tant que flux. Tout d'abord, un programme que nous appelons *vertex programme* est appliqué sur tous les vertex d'entrée. Ce traitement correspond sur la fig. 3.18, page 52, au bloc *vertex processor*, aussi appelé par le terme anglais *vertex shader*. Selon le paradigme stream de traitement des données, le même programme est appliqué sur tous les vertex. Ce programme réagit localement et son exécution sur un vertex est donc indépendante de l'exécution sur d'autres vertex. Il dispose des paramètres globaux pour la lecture mais il ne peut pas les modifier et il ne peut écrire non plus de valeur quelconque dans une mémoire commune. En effet, il ne peut que modifier les valeurs des propriétés du vertex qu'il traite. Il est donc impossible de créer de nouveaux vertex, d'avoir une variable globale partagée modifiable par tous les vertex ou d'influencer l'exécution d'autres vertex. En revanche, nous pouvons accéder aux données globales stockées dans la mémoire comme textures en utilisant le processus d'*échantillonnage des textures*.

Les vertex traités passent le long du pipeline dans l'unité suivante, l'unité d'*assemblage des primitives*. Cette unité se charge de l'extraction des primitives graphiques de base (décrites par la commande graphique) et de tous leurs descripteurs à partir du flux des vertex. Il s'agit, en effet, du décodage des formes de base à partir de la commande graphique que nous avons passée au GPU. Cette unité prépare les primitives géométriques pour l'étape suivante du pipeline – la digitalisation de cette forme.

La digitalisation est décrite sur la fig. 3.18 comme bloc *Rastérisation et interpolation*, qui se charge de la création des *fragments*. Les fragments sont la représentation digitale d'une forme dans un espace *3D* digital. La façon exacte dont on dérive les fragments à partir de cette forme est dépendante des paramètres du modèle de projection que nous avons choisi et des dimensions de notre framebuffer que nous pouvons imaginer, d'une manière simplifiée, comme une surface digitale destinée au rendu de notre modèle 3D.

Soulignons la différence entre les fragments et les pixels. Les pixels sont les points de l'espace digital 2D dont la valeur est la couleur. Les fragments sont des structures beaucoup plus riches dans leur contenu, on qualifie un fragment comme un prototype d'un pixel. La première différence est dans le fait qu'un fragment est décrit par les coordonnées 3D ; simplement dit, un fragment possède d'une information sur la profondeur. Cela donne la possibilité d'avoir dans le pipeline plusieurs fragments de la même position 2D qui diffèrent dans leur profondeur. Et surtout, un fragment peut contenir davantage d'informations qu'un pixel. En revanche, un pixel nous sert comme une structure pour les données qui sont inscrites à la fin du traitement dans le framebuffer, il contient une position 2D fixée et n'a plus d'information sur la profondeur. Dans les algorithmes classiques de synthèse des images, il n'y a qu'un pixel qui est généré pour une coordonnée donnée[1].

Les informations que nous plaçons dans les fragments sont obtenues à partir des vertex. Puisqu'une forme géométrique est décrite, dans la plupart des cas, par moins de vertex qu'il n'y a de fragments dérivés de cette forme, nous sommes obligés de distribuer l'information des vertex vers les fragments. On utilise avec succès l'interpolation des valeurs des paramètres à partir de plusieurs vertex (2, 3, voir plus) pour cette distribution. Cette approche est largement utilisée pour obtenir la position d'un fragment

[1] Avec l'exception d'un traitement des points comme des *point-sprites* qui peuvent générer plusieurs fragments et ensuite plusieurs pixels pour une seule coordonnée 2D

à l'intérieur d'une forme, pour interpoler les couleurs, les vecteurs décrivant les propriétés de la surface, les coordonnées des textures et peut même être utilisée pour interpoler d'autres données qui n'ont pas un correspondant direct dans le graphisme 3D.

Après la création et l'obtention des paramètres interpolés des vertex, les fragments passent à l'unité de traitement des fragments décrite sur la fig. 3.18 par le bloc *Fragment processeur* et appelée également par le terme anglais *fragment shader*. Il s'agit d'une unité de traitement qui traite le flux de la même manière que le *Vertex processeur*. C'est-à-dire que nous pouvons accéder aux paramètres globaux et même échantillonner les textures mais nous ne pouvons pas influencer le traitement d'autres fragments ou créer de nouveaux fragments. Dans les applications graphiques 3D, cette unité est utilisée pour un calcul intensif de la couleur locale du fragment et implémente la partie principale du calcul de l'illumination de la partie de la surface 3D qui correspond au fragment. C'est pourquoi ce bloc fonctionnel est fortement parallélisé, plus que celui du processeur des vertex, et son implémentation matérielle est optimisée pour l'accès massif aux textures, ce qui se traduit par une structure particulière des mémoires caches qui ont pour but d'exploiter au maximum la localité des données lors d'échantillonnage des textures.

Les fragments traités passent dans l'unité suivante que nous désignons comme unité de post-traitement qui correspond dans la figure 3.18 au bloc *Raster opération* et qui regroupe tout un tas de fonctions utiles. Parmi elles, nous trouvons, selon les capacités de notre GPU cible, le fusionnement (appelé *blending* avec les valeurs du framebuffer), application d'une transformation fixe sur les valeurs de la couleur, les opérations du masque, les tests sur les valeurs, la convolution, la fonction *stencil* de masquage conditionnel, suppression des fragments selon la valeur de leur profondeur, accumulation des valeurs. Pour plus d'information, nous renvoyons le lecteur à la littérature [Gra03, WNDS99].

Le flux des pixels qui sortent de ce bloc est écrit continuellement dans le framebuffer. Le framebuffer a aujourd'hui de larges possibilité qui ne se restreignent pas seulement à un vidéobuffer. En effet, il est possible d'y connecter diverses zones de mémoire d'un GPU et obtenir ainsi la sortie des résultats qui est configurable par l'utilisateur lors de l'exécution du programme dans les GPP.

Pour conclure, nous devrions noter que l'évolution de la structure du pipeline graphique est constante. Il est fort probable que les fonctionnalités dont nous disposons à présent seront élargies, comme c'était déjà le cas à plusieurs reprises. Par exemple, avant 2004, nous n'avions pas encore la possibilité d'accéder aux textures à partir des *Vertex programmes*. En même temps, le schéma de blocs que nous avons présenté sur la fig. 3.18, page 52, est assez général et devrait être valable dans sa forme encore un certain temps. Cette remarque est due au fait que la recherche et le développement dans ce domaine sont très intensifs et nous pouvons nous attendre à des changements plus ou moins importants. Comme exemple d'une implémentation non standard du schéma classique du pipeline graphique, nous pouvons citer l'architecture ATTILA [MGR+05a, MGR+05b] dans laquelle les unités de traitement des vertex et des fragments sont réunies dans des blocs fonctionnels universels et programmables. Cette architecture distribue les moyens matériels pour le calcul en stream selon l'intensité du traitement dans un bloc ou l'autre et donne aux unités universelles soit la fonction d'un *vertex processeur*, soit d'un *fragment processeur*. Il s'agit d'une architecture tout-à-fait intéressante pour notre traitement dans lequel nous utilisons largement plus les unités des fragments que celles des vertex et où il serait avantageux de disposer de maximum de moyens matériels dédiés au traitement des fragments, même temporairement, pour ainsi obtenir une parallélisation encore plus forte du traitement en flux.

3.4.5.4 Interfaces de programmation des GPP - DirectX et (Open)GL

Il faut disposer d'outils qui permettent aux programmeurs des applications sur les GPP de facilement utiliser les fonctionnalités matérielles des GPU. Deux groupes des API sont actuellement disponibles. Le premier groupe est constitué par DirectX de Microsoft. Le deuxième groupe est constitué par OpenGL de Silicon Graphics Incorporated et par d'autres dont la syntaxe et les fonctionnalités sont très proches d'OpenGL et dont nous pouvons citer Mesa ou l'API OpenGL ES destiné pour les systèmes embarqués.

Les deux offrent une interface qui permet de manier les GPU mais elles diffèrent dans la philosophie

de leur construction. Tandis que DirectX est lié aux technologies de Microsoft, OpenGL est modulable par extensions où chaque fabricant des architectures de traitement 3D peut exposer les fonctionnalités de son matériel par une extension propriétaire. La standardisation de la structure de base du OpenGL est effectuée par *OpenGL Architectural Review Board*, ARB, qui regroupe les équipes issues de grands fabricants de solutions pour le graphisme. Les extensions du standard OpenGL qui ont été approuvées par ARB mais qui n'étaient pas, pour différentes raisons, incluses au standard sont appelées ARB Extensions et exportent le plus souvent l'interface des fonctionnalités des GPU les plus récents.

Les solutions logiciels basées sur OpenGL (ou Mesa) ont un grand avantage par rapport au DirectX – c'est leur facilité de portage vers une plateforme de système d'exploitation différente. Nous avons utilisé pour nos expériences les deux plateformes pour finalement migrer totalement vers Mesa car nous voulions garder la portabilité de nos solutions et effectuer des tests comparatifs pour des systèmes d'exploitation différents (Windows XP et une distribution de Linux).

Un autre avantage d'OpenGL pourrait également être mentionné. C'est le fait que la plupart des langages de script (e.g. Python, Haskell) ont la possibilité de programmation des GPU via une bibliothèque compatible avec OpenGL même si les fonctions qu'ils offrent n'incluent pas toujours les ARB Extension dont nous aurions besoin pour notre travail. Pour plus de détails sur ces interfaces, nous renvoyons le lecteur à la littérature traitant d'OpenGL [WNDS99, HAL04, Dal03, KBR03] et à la littérature traitant de DirectX [Gra03, Mig04].

3.4.5.5 Langages d'ombrage pour le temps réel - Cg, GLSL, HLSL

Comme les processeurs graphiques ont évolué à partir des pipelines fixes ou configurables vers les pipelines programmables, les langages de haut niveau destinés aux blocs programmables à l'intérieur d'un GPU ont fait leur apparition. On les appelle les *langages d'ombrage pour le temps réel* (angl. *Shading languages*), avec une spécification du temps réel dans leur nom pour les distinguer des langages d'ombrage (classiques) destinés au rendu des images et des scènes complexes et où l'exigence du temps réel ne figure pas (e.g. RenderMan de Pixar Animation Studios). Leur nom *ombrage* (angl. *shaders*) est dérivé du fait que le traitement des vertex et des fragments est utilisé pour le calcul des ombres ou de l'illumination locale dans la synthèse des images.

Le premier des langages d'ombrage pour le temps réel que nous mentionnons est nommé Cg [FK03]. Son nom est une abréviation de *C for graphics* et il s'agit d'un langage développé par NVidia. Il s'incorpore à travers des bibliothèques fournies par le fabricant dans les deux API présentés. Ainsi, nous pouvons l'utiliser avec le DirectX et aussi avec OpenGL. Le deuxième langage d'ombrage est nommé HLSL (*High Level Shading Language*) de Microsoft et vient de compléter DirectX. Le troisième est nommé GLSL (*OpenGL Shading Language*) et complète OpenGL via les ARB Extensions.

Tous les trois sont largement inspirés du langage C, dont ils ont la syntaxe, mais ils ont défini leurs propres types et identificateurs qui reflètent les types utilisés dans le traitement sur les GPU. Ainsi, il n'est pas difficile pour un programmeur connaissant le langage C d'adopter ces langages. Pour une comparaison des fonctionnalités de ces trois langages, nous renvoyons le lecteur à un article [Lov05].

Ces langages nous servent comme base pour le traitement à l'intérieur du GPU. La grande partie de l'algorithme de traitement d'images sur les GPU est implémentée par ces langages car c'est le GPU qui assure l'exécution tant que le programme d'application, exécuté dans le GPP, ne fait que coordonner le travail du GPU, assure l'envoi des commandes graphiques et se charge de la lecture des résultats.

3.4.6 Calcul sur les flux de données avec les GPU

3.4.6.1 GPGPU

Notre manière de travailler entre dans la catégorie du *calcul général sur les processeurs graphiques* qui s'est différenciée du calcul classique de rendu d'images 3D et constitue aujourd'hui une branche

parmi les utilisateurs des GPU. Elle est représentée par une communauté scientifique grandissante appelée GPGPU [GPG], ce qui est une abréviation du terme anglais *General Processing on Graphics Processing Units*.

Le calcul général sur les processeurs graphiques a certaines propriétés intéressantes. Il se présente d'un côté par l'uniformité du calcul, propre au paradigme stream, qui le relie avec d'autres architectures à flux de données. D'un autre côté, il présente la propriété du parallélisme spatial qui le distancie des architectures classiques à flux de données et qui le valorise par rapport à ces dernières. Le parallélisme spatial est bien incorporé dans l'architecture du pipeline graphique car la notion de la position est présente dans les deux structures de données principales des GPU qui sont traitées en flux – vertex et fragments. Ainsi, les kernels de traitement du flux ont implicitement à disposition la notion de la position, ce qui fait des vertex et des fragments des tokens tout-à-fait intéressants pour le traitement de plusieurs dimensions et plus particulièrement pour le traitement des images dans notre cas.

Le traitement général sur les architectures spécialisées pour le graphisme n'est pas une idée nouvelle[1]. Diverses architectures ont été proposées et construites, commençant probablement en 1978 par Ikonas [Eng78] et continuant progressivement jusqu'à nos jours. En 1988, Fournier et Fussell ont pointé dans leur article [FF88] les avantages mais aussi les désavantages de l'utilisation des GPU pour un calcul algorithmique, différent de celui destiné à la visualisation. Ils ont modélisé le framebuffer par un modèle stream et ont cherché, sur les problèmes précis (le problème de visibilité des facettes et le problème du calcul des ombres) comment la puissance de ces traitements va augmenter si on ajoutait à ce modèle plus de capacités de calcul. L'arrivée des processeurs graphiques programmables a largement changé le point de vue sur les GPU : ceci se reflète dans leur utilisation de plus en plus fréquente pour les tâches d'un calcul massif. De nos jours, on essaie d'exploiter les capacités particulières concernant surtout la bande passante large entre le processeur graphique et sa mémoire dédiée (graphique) mais également d'exploiter les possibilités du calcul en virgule flottante. Un excellent article [OLG+05] analyse la bibliographie des réalisations scientifiques sur les processeurs graphiques faites jusqu'à l'année 2005. D'autres articles [CDPS03, GWH05, TC05] présentent une introduction à la problématique du traitement général sur les GPU.

Un projet logiciel intéressant appelé Brook [BFH+04a, BFH+04b] dédié au calcul stream sur les GPU qui explore leurs capacités d'évaluation massive en virgule flottante a été conçu par l'équipe scientifique de l'université de Stanford. Il propose un langage spécialisé qui élargit la syntaxe du langage C et introduit de nouvelles structures pour les streams et les kernels. Ainsi, il permet de traiter les donnés en flux sans que son utilisateur n'ait besoin de connaître l'implémentation exacte de son programme dans les termes de la programmation d'une application de graphisme 3D à l'aide d'un API et d'un langage d'ombrage quelconque. C'est, en effet, le travail de compilateur brcc du Brook qui traduit le code travaillant avec les streams en un code d'application 3D qui inclut les définitions des shaders appropriés au traitement. L'environnement d'exécution run-time brt du Brook se charge ensuite de la configuration du pipeline graphique pour ce traitement, de l'envoi des données dans les GPU, de l'application des programmes appropriés sur ces données et de la récupération des résultats.

Le grand désavantage de Brook pour notre travail en morphologie mathématique est le fait qu'il n'intègre pas encore le traitement stream avec les types de nombres entiers et qu'il n'implémente qu'un certain nombre d'opérations de base sur les streams. Le manque de types entiers est dû au fait que les GPU ont été conçus prioritairement pour le calcul en virgule flottante et que Brook ne fait qu'exploiter les capacités de cette orientation particulière. Ce n'est pas le fait d'utiliser les types *floating point* de 32 bits pour effectuer les évaluations qui défavorise cette approche pour le calcul morphologique, mais le coût des transferts de nos données codées dans ces types vers et depuis le GPU. Sachant que ce n'est pas la puissance des GPU dans le calcul en virgule flottante que nous utilisons mais la largeur de la bande passante à la mémoire des données qui surpasse celle des CPU classiques qui nous intéresse le plus. Dans ce cas, il serait plus intéressant d'avoir dans Brook la possibilité de travailler aussi avec les types *integer* de 8 bits ce qui n'est pas, pour l'instant, le cas.

[1] Pour plus d'informations historiques, nous renvoyons le lecteur aux articles [HCSL02, Ven03]

Addition des éléments correspondant de 2 streams utilisant Brook taille des streams = 1000, type des données = float 32 bit				
Matériel	Avec transfert GPP↔GPU		Sans transfert (uniquement calcul)	
	temps ms	taux d'accélération	temps ms	taux d'accélération
GPP	0.75	1.0	0.73	1.0
GPU OpenGL	0.19	3.9	0.09	8.1
GPU DirectX9	0.19	3.9	0.09	8.1

GPP = Intel Pentium 4 à 2.4 GHz single thread ; GPU = NVidia GeForce 6800 LE sur AGP4x. Exécuté en 5 séries de 1000 itérations, le temps présenté est le temps moyen pour 1 itération de la série la plus favorable.

TAB. 3.3 : Résultats expérimentaux de l'opération addition sur les streams utilisant Brook pour l'exécution dans le GPU

Pour illustrer ce point, regardons la tab. 3.3 où nous présentons les résultats d'expérimentation avec Brook. L'opération effectuée est une simple addition de deux vecteurs de 1000 éléments dont le type est *float* de 32 bits. Les temps qui sont présentés dans ce tableau sont intéressants pour un calcul en virgule flottante car ici, nous obtenons un facteur d'accélération 3.9 si nous comparons l'exécution sur le GPU NVidia GeForce 6800 LE avec l'exécution plus lente sur le GPP Intel Pentium 4 à 2.4 GHz, single thread.

Il est, en effet, possible d'utiliser Brook pour le calcul morphologique en adoptant son style de travail et les outils de développement. Mais il faut noter que pour pouvoir obtenir des temps d'exécution encore plus avantageux et notamment dans les algorithmes de la morphologie mathématique qui travaillent avec les nombres entiers, il faut effectuer un travail différent de celui que Brook nous propose. Malgré cela, Brook reste un outil de développement intéressant pour le traitement en stream et spécialement si nous voulons être abstraits de l'architecture particulière des GPU.

3.4.6.2 Calcul de la synthèse des images pour la morphologique mathématique sur les GPU

Le traitement morphologique des images 2D que nous voulons effectuer avec le pipeline graphique sur les GPU nous dicte précisément quel modèle mathématique nous allons utiliser. Nous n'allons pas utiliser la projection perspective de l'espace 3D à 2D car elle n'a pas d'utilité pour notre traitement. En revanche, nous allons utiliser la projection orthogonale qui constitue un des modes standard de traitement sur les GPU. Également, les modèles géométriques que nous emploierons ne seront pas complexes et nous nous contenterons des listes de certaines primitives géométriques, notamment *point*, *ligne*, *rectangle*, que nous allons utiliser comme les commandes graphiques pour lancer le calcul sur les zones de l'image.

Ce fait est traduit dans notre traitement par l'absence d'utilisation explicite des triangles, largement utiles dans le graphisme 3D. Les triangles sont utilisés, en effet, pour la simplification des rectangles à des formes géométriques de base où chaque rectangle est exprimé par deux triangles non recouvrants.

Cette page est blanche par intention

CHAPITRE 4

Formalisme fonctionnel adopté pour la morphologie mathématique

4.1 Approche fonctionnelle et impérative

Il existe deux façons principales pour décrire un algorithme pour une machine qui sont différentes et opposées. La première consiste en l'utilisation des langages fonctionnels, la deuxième en l'utilisation des langages impératifs.

Les langages fonctionnels ont leurs racines dans le Lambda calcul [1]. Le Lambda calcul[BB94, Wik05b] est un modèle du calcul, une formalisation de la notion du calcul, basée sur les fonctions et sur la normalisation d'une expression calculée par la réduction et qui a pour but obtenir une forme normale, le résultat du calcul. C'est-à-dire tout ce que l'on fait habituellement en mathématique pour calculer un résultat en utilisant une formule donnée.

Les langages impératifs se basent sur le changement d'état du programme. Dans ces langages, le programme est décrit comme une séquence d'instructions. Les machines du calcul physiquement construites sont décrites par un automate fini et par conséquent, la séquence des instructions pour ces machines est également un programme dans un langage impératif (langage machine). Le modèle du calcul pour ces langages est la machine de Turing, c'est-à-dire une machine séquentielle.

On doit également souligner que les deux modèles sont équivalents et on les appelle *Turing-équivalents* car c'était Turing qui a prouvé[Tur37] que les deux modèles désignent la même classe des fonctions calculables. On peut simuler la normalisation des termes lambda par la machine de Turing. Mais on peut simuler également la machine de Turing par les termes lambda. Et comme il est dit dans la thèse de Church-Turing[Wik05d], tout algorithme peut être calculé par une machine de Turing, donc par équivalence tout algorithme peut être calculé par le Lambda calcul.

Nous introduisons le formalisme fonctionnel dans cette thèse pour pouvoir être abstrait dans la description des algorithmes. En décrivant seulement les fonctionnalités, nous ne sommes pas obligés de nous restreindre à un type d'architecture donnée et d'utiliser un langage impératif quelconque. L'article intitulé *Can programming be liberated from von Neumann style*[Bac77] discute cette problématique en démontrant les points négatifs de l'approche impérative à la programmation et propose comme solution la programmation algébraïque représentée par l'approche fonctionnelle à la programmation.

Le Lambda calcul dans sa forme théorique est peu utilisé dans la pratique où l'on préfère travailler avec un des langages dérivés du Lambda calcul – un des langages fonctionnels. Les langages fonctionnels[2] implémentent le Lambda calcul. Ce sont les langages de programmation de haut niveau qui n'uti-

[1] Souvent écrit en utilisant le caractère grec comme λ-calcul. Nous n'utilisons pas cette notation.

[2] Comme langages fonctionnels nous pouvons citer Miranda et Haskell ou même LISP et ML, les deux derniers considérés comme fonctionnels impératifs car permettant d'exprimer aussi les constructions impératives

lisent pas la machine d'état pour décrire un programme comme c'est le cas chez les langages impératifs[1] mais ils préfèrent exprimer le programme à l'aide des fonctions qui prennent des paramètres d'entrée et qui ne retournent qu'un paramètre de sortie à la fois. L'utilisation des types et des structures de haut niveau est propre à ces langages. Ils permettent d'exprimer d'une façon simple les constructions complexes qui prendraient plusieurs lignes dans un langage impératif. Ces constructions prennent souvent beaucoup moins de lignes ou seulement une seule dans un langage fonctionnel. Par exemple, l'application d'une fonction à chaque élément d'une liste prend une ligne et utilise la récursion.

4.2 Haskell et les bases des langages fonctionnels

4.2.1 Syntaxe du Haskell

Nous avons choisi la syntaxe du langage Haskell pour exprimer les algorithmes dans cette thèse. Le nom de ce langage, Haskell, est en honneur de Haskell Brooks Curry [Wik06g] dont les travaux dans la logique mathématique servent comme bases aux langages fonctionnels. Il s'agit d'un langage fonctionnel qui émane du Lambda calcul, qui est bien défini dans sa version actuelle nommée Haskell98 [Jon03] mais qui est également bien vivant car les travaux de définition de son successeur se poursuivent ; il est utilisé avec succès dans la recherche y compris par les membres[2] de Microsoft Research [HMJH05], par exemple.

Notre choix du Haskell a été le résultat d'une petite recherche que nous avons menée et dont le but était de choisir un formalisme satisfaisant un certain nombre de desiderata. Dans notre cas, c'est Haskell qui a contenté nos exigences par la combinaison des propriétés suivantes :

- les description implicitement formelle des algorithmes à travers Haskell,
- la proximité de la notation utilisée en mathématiques où même les lecteurs non instruits peuvent facilement comprendre les algorithmes exprimés en Haskell,
- les capacités larges de modélisation qui sont adaptées à nos besoins,
- pouvoir vérifier la bonne construction d'une définition en utilisant le *parser* pour l'analyse de syntaxe,
- la possibilité de simulation par l'exécution d'un programme, ce qui nous permet de vérifier le bon fonctionnement sur les données réelles.

La combinaison de ces propriétés nous a mené par la suite à l'utilisation du Haskell pour notre travail et pour nos explications.

Pourtant, nous ne sommes pas les seuls de se poser de telles exigeances et de choisir Haskell comme un outil ; il y a d'autres chercheurs qui l'utilisent pour la description mathématique, cf. l'article électronique intitulé *Eleven Reasons to use Haskell as a Mathematician* [Unk06].

Le Lambda calcul [BB94] travaille avec deux opérations de base, l'*application* et l'*abstraction*. L'opération d'application qui est exprimée dans le Lambda calcul par

$$A \cdot E$$

ou également par plus court

$$A\ E$$

indique l'application d'un algorithme, perçu par A, sur une entrée, perçue par E. L'application dans Haskell correspond, si elle est explicitement mentionnée, à la fonction $ utilisée comme

```
A$E
```

mais nous pouvons aussi employer la notation plus courte, de la même façon que dans le Lambda calcul. Cette notation omet la fonction d'application explicite et est utilisée comme :

```
A E
```

[1] Comme langages impératifs nous pouvons citer Basic, Pascal, C, C++, Java, et d'autres.

[2] Citons les travaux de Simon Peyton Jones portant sur les utilisations du Haskell dans les technologies de Microsoft

La deuxième opération de base du Lambda calcul est l'*abstraction*. Si $M = M[x]$ est une expression qui contient x (ou autrement dit dépend de x), puis $\lambda x.M[x]$ désigne la fonction $x \mapsto M[x]$. Par exemple, $\lambda x.x * x$ définit la puissance de 2. Dans Haskell, nous avons également la possibilité de définir une fonction par le terme λ comme :

```
λx→x*x
```

Cette expression définit une nouvelle fonction sur place. Elle peut être utilisée dans les définitions des fonctions dans Haskell, comme nous pourrons le voir par la suite, par exemple, dans la définition de la fonction \$, page 62.

Une des spécialités du Lambda calcul est la façon d'exprimer les fonctions de plusieurs arguments par itération de l'application[1] et nous parlons ainsi d'une succession des *applications partielles* [Wik06c]. Ainsi, si $f(x, y)$ dépend de deux arguments x et y, nous pouvons définir dans le Lambda calcul $F_x = \lambda y.f(x, y)$ qui ne dépend que de y et $F = \lambda x.F_x$ qui ne dépend que de x et nous obtenons par conséquent $(F\ x)y = F_x\ y = f(x, y)$. Haskell suit cette logique et les fonctions de plusieurs arguments sont définies exactement de cette manière même si la syntaxe du Haskell le cache quelquefois. Par exemple [HPF99], la fonction add d'addition de deux arguments est définie dans Haskell comme :

```
add (x,y) = x+y
```

ce qui est un exemple d'une définition *uncurried*. Cette définition est équivalente à une définition *curried* :

```
add x y = x+y
```

qui correspond à

```
add = λ x y → x+y
```

ce qui est une notation propre à Haskell mais qui n'exprime rien d'autre que

```
add = λ x → λy → x+y
```

La dernière définition correspond dans la notation du Lambda calcul à l'expression $\lambda x.\lambda y.x + y$ qui peut être récrite comme $\lambda x.(\lambda y.x + y) = \lambda x.F_x$ en utilisant les fonctions partielles comme décrit auparavant. Les fonctions standards **curry** et **uncurry** du Haskell sont dédiées à ce travail.

Avant de passer à nos propres définitions et expressions dans Haskell, nous voudrions introduire certaines notions de base pour que le lecteur non familier avec Haskell ou avec d'autres langages fonctionnels puisse poursuivre notre raisonnement et comprendre la syntaxe et la façon de construire les algorithmes décrits par la suite. Pour plus d'informations, nous orientons le lecteur vers une brève introduction du Haskell98 [HPF99].

La fonction de l'identité polymorphe id est un exemple trivial d'une fonction définie par Haskell. Elle va nous servir pour expliquer la syntaxe de ce langage :

```
id    :: α → α
id x  =  x
```

La première ligne définit, par le symbole ::, la signature du type de la fonction **id** pour ses paramètres et la valeur de retour. Nous lisons la définition de gauche à droite, dans le sens des flèches. Dans ce cas, il s'agit d'une fonction qui prend un paramètre d'un type polymorphe α, et transforme sa valeur en une autre valeur du même type α.

La deuxième ligne définit, en utilisant le symbole =, le corps de la fonction. Notons que les langages basés sur le Lambda calcul définissent les fonctions comme les règles de réécriture. En définissant une fonction dans Haskell, nous prescrivons, en effet, cette règle de réécriture. L'expression à gauche de = qui correspond à la signature de la fonction sera récrite par l'expression à droite de =.

Dans ce cas précis de la fonction *id*, nous avons à gauche = la signature de la fonction avec un paramètre *x* qui sera récrite par l'expression à droite de =, c'est-à-dire en *x* sans le changer. C'est la définition d'identité valable pour les valeurs de n'importe quel type.

[1] Il s'agit d'une idée due à M. Schönfinkel, appelée également *curryfication* [Wik06c] (du terme anglais *currying*), selon le nom de Haskell Brooks Curry qui a également et indépendamment introduit cette idée [BB94]

La code source de l'identité polymorphe en Haskell est le suivant :

```
id   :: a -> a
id x = x
```

Le côté pratique de notre approche réside dans la possibilité de vérifier automatiquement la syntaxe et le fonctionnement des définitions en utilisant le compilateur GHC[GHC] du Haskell, car toutes les définitions sont également les fonctions définies dans ce langage.

4.2.2 Fonctions de base du Haskell

Les définitions des fonctions que nous décrivons à cette place appartiennent à la base de la programmation fonctionnelle et sont incluses dans Haskell. Vu qu'elles se trouvent abondamment employées dans nos prochaines descriptions, nous tenons à présenter au lecteur leurs définitions exactes.

: est le constructeur d'une liste qui travaille avec un élément et une autre liste comme les arguments. Par exemple, l'expression $a : []$ définit la liste $[a]$ avec un seul élément a. Les listes contenant plusieurs éléments peuvent être construites par l'applications successives de ces constructeurs, e.g. $a : b : c : []$ construit la liste $[a, b, c]$.

$ est un opérateur d'application qui applique une fonction sur un paramètre. Il est utilisé dans une écriture dense ou là où l'on veut explicitement insister sur l'application de la fonction sur un paramètre :

$$\begin{aligned}&(\$) \quad :: \quad (\alpha \to \beta) \to \alpha \to \beta\\ &f\,\$ \quad = \quad \lambda x \to f(x)\end{aligned}$$

$\circ$ est un infixe de composition, il crée à partir de deux fonctions une seule fonction composée :

$$\begin{aligned}&(\circ) \quad :: \quad (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma\\ &f \circ g \quad = \quad \lambda x \to f(g(x))\end{aligned}$$

$\setminus\!\setminus$ est un infixe de différence de deux listes, e.g. l'expression $[1, 2, 3, 4] \setminus\!\setminus [2, 4]$ donne comme résultat une liste $[1, 3]$.

$+\!\!+$ est un infixe de la concaténation de deux listes, e.g. l'expression $[1, 2] +\!\!+ [3, 4]$ donne comme résultat une liste $[1, 2, 3, 4]$.

map applique une fonction à chaque élément d'une liste. Nous pouvons y percevoir le calcul sur les streams et faire ainsi un lien avec le kernel d'application que nous avons décrit dans 3.4.1.1, page 45. Sa définition utilise la récursion :

$$\begin{aligned}&\textbf{map} \quad :: \quad (\alpha \to \beta) \to [\alpha] \to [\beta]\\ &\textbf{map}\ f\ [\,] \quad = [\,]\\ &\textbf{map}\ f\ (x{:}xs) = f\ x\ :\ (\textbf{map}\ f\ xs)\end{aligned}$$

foldr1, foldl1 La fonction foldr1 prend les deux derniers éléments d'une liste et applique sur eux une fonction. Puis elle applique cette même fonction entre le résultat obtenu et le troisième élément à partir de la fin de la liste. Et ainsi de suite juqsu'à ce que tous les éléments soit traités. Nous pouvons y percevoir le calcul sur les streams et faire ainsi un lien avec le kernel de réduction que nous avons décrit dans 3.4.1.2, page 45 :

$$\begin{aligned}&\textbf{foldr1} \quad :: \quad (\alpha \to \alpha \to \alpha) \to [\alpha] \to \alpha\\ &\textbf{foldr1}\ f\ [x] \quad = \quad x\\ &\textbf{foldr1}\ f\ (x{:}xs) = f\ x\ (\textbf{foldr1}\ f\ xs)\end{aligned}$$

Une autre fonction, foldl1, travaille semblablement mais elle commence le traitement à partir du début de la liste et progresse vers l'arrière.

foldr, foldl La fonction foldr est une autre fonction de réduction d'un stream. Se différenciant de la précédente, foldr1, elle utilise une valeur d'entrée d'un autre type que celui du stream. Ainsi, elle permet d'effectuer la réduction sur un stream d'une manière plus générale. Elle commence par l'application de la fonction f sur la valeur d'entrée et réutilise le résultat itérativement sur tous les autres éléments de la liste.

```
foldr              :: (α → β → β) → β → [α] → β
foldr f z []       = z
foldr f z (x:xs)   = f x (foldr f z xs)
```

L'utilité de cette fonction est évidente. Un exemple que nous pouvons donner pour illustrer les cas d'utilisation de cette fonction peut être le calcul d'un histogramme où nous voulons obtenir, à partir d'une liste des éléments triviaux, tels que des numéros, une structure plus complexe, tel que l'histogramme.

La fonction foldl travaille pareillement, mais elle exécute la réduction à partir de la fin de la liste et progresse vers l'avant.

filter retourne la liste avec les éléments qui satisfont le critère p. Nous pouvons y percevoir le calcul sur les streams et faire ainsi un lien avec le kernel de filtrage que nous avons décrit dans 3.4.1.3, page 45 :

```
filter             :: (α → Bool) → [α] → [α]
filter p []        = []
filter p (x:xs)  | p x          = x : filter p xs
                 | otherwise    = filter p xs
```

zip crée, à partir de deux listes, une liste des tuples où chacun des tuples contient, respectivement, un élément de la première liste et un élément de la deuxième, e.g. zip $[1,2,3]$ $[4,5,6]$ a pour résultat $[(1,4),(2,5),(3,6)]$:

```
zip                :: [α] → [β] → [(α,β)]
zip (x:xs) (y:ys)  = (x,y) : zip xs ys
zip _ _            = []
```

fst retourne le premier élément d'un tuple :

```
fst          :: (α,β) → α
fst (x,_) = x
```

snd retourne le deuxième élément d'un tuple :

```
snd          :: (α,β) → β
snd (_,y) = y
```

4.3 Primitives de stockage et de représentation des données

Nous définissons de nouveaux types pour le stockage et la représentation des données en nous appuyant sur les types de base du Haskell. Pour la plupart, nous n'allons utiliser que la redéfinition des identificateurs des types déjà existants afin d'introduire au type l'information sémantique qui nous servira à être clair et compréhensible dans les définitions complexes. Notons que les identificateurs de type dans Haskell doivent commencer par une lettre majuscule.

Deux mot clés du Haskell sont dédiés à ce but, **data** et **type**. Le mot clé **data** qui définit un véritable nouveau type de données, e.g. la définition

```
data AB = A | B
```

définit le type énumératif avec deux identificateurs possibles – A et B. Le mot clé **type** définit un nouvel identificateur du type en se basant sur les types déjà existants ou sur leur combinaison. Par exemple, l'expression

```
type B = A
```

définit un nouvel identificateur B comme étant le même type que l'identificateur A. Ou encore, l'expression

```
type B = (A, A)
```

définit un nouvel identificateur B comme un tuple dont les deux éléments sont du type A.

4.3.1 Types de base

4.3.1.1 Indexation

Les numéros entiers seront utilisés pour exprimer les indices lors de l'indexation des éléments des arrays ou des listes. Nous pourrions utiliser directement les valeurs d'un type de Haskell, **Int**, mais nous préférons redéfinir son identificateur en une forme plus courte, I, qui va s'avérer très utile dans les définitions plus longues.

```
type I = Int
```

4.3.1.2 Flux de données

La structure présente dans Haskell et convenable pour le stockage et le travail avec les données en stream (sur les flux de données) est celle d'une liste. La liste dans Haskell est définie, en utilisant la classe *list*, par le constructeur []. La liste est polymorphe et elle ne peut contenir que des éléments du même type. Par exemple, l'expression [] définit une liste vide et l'expression

```
xs = ['x','y','z']
```

définit une liste xs dont les éléments sont des caractères, le premier élément a la valeur 'x', le deuxième la valeur 'y' et le troisième la valeur 'z'.

L'indexation des éléments dans une liste est effectuée par !! (double point d'exclamation) et commence à partir de 0. Ainsi, le premier élément détient l'index 0, le deuxième 1, etc. Par exemple, l'expression

```
xs!!1
```

rendra la valeur du deuxième élément, c'est-à-dire 'y'.

La deuxième manière de construire une liste est celle que nous avons déjà mentionnée (page 62) et qui utilise le constructeur : (deux-points). Celui est très utile lors de la composition des listes à partir des éléments, e.g. l'expression

```
ys = 1:2:3:[]
```

crée une liste ys qui est égale à $[1, 2, 3]$.

La liste est un type essentiel dans les langages fonctionnels, Haskell inclus. Ainsi, les fonctions sur les listes sont incorporées dans *Prelude*, le cœur du Haskell. Pour plus de précision nous renvoyons le lecteur vers le manuel[Jon03] du langage Haskell. Nous allons utiliser les listes par la suite dans nos algorithmes pour exprimer explicitement la notion d'un stream et pour ordonner des données lors d'un traitement en flux.

4.3.1.3 Types de données paquetées pour le traitement SIMD

Le traitement des données sur une architecture SIMD est un traitement en parallèle. Ainsi, il faut avoir des structures de données qui seraient convenables à un tel traitement. Pour le traitement stream sur les processeurs avec les capacités SIMD, nous utilisons des blocs de données élémentaires du même type regroupés dans un type composé qui constitue la base pour notre travail. Nous parlons ainsi des types de *données paquetées*. Ces types sont appelés dans la littérature également les *données vectorielles*.

Même si dans nos traitements nous pouvons rencontrer les vraies données vectorielles comme c'est le cas chez les coordonnés x, y, z, w sur les cartes graphiques, dans la plus grande partie de nos algorithmes, les données que nous traitons ne correspondent pas à un vrai vecteur d'un point de vue mathématique mais plutôt à des groupes de pixels. C'est la raison pourquoi nous préférons garder le terme *données paquetées* pour nos propos car nous pensons que l'appellation "*paqueté*" reflète mieux la réalité. De plus, elle correspond à un terme que nous trouvons dans la littérature anglaise – "*packed-vector*".

En théorie, nous pouvons choisir une taille arbitraire du type de données paquetées, en pratique nous sommes restreints par le choix de l'architecture finale et par ses capacités. La plupart des architectures contemporaines possèdent des types paquetés de 4 à 16 éléments (cf. les technologies multimédia, tab. 1.2, page 21, dans le chapitre Motivation). Sur les architectures spéciales et dédiées nous pouvons trouver un nombre plus élevé d'éléments dans un type paqueté (de 64 à 256), e.g. prototype du processeur dynamiquement reconfigurable *Vision chip* [KKI04] ou le système *IMAP-VISION* [PJ00].

Un exemple du type paqueté *iu8vec8* (notation MorphoMedia [Bra05]) regroupant 8 éléments unsigned integer de 8 bit est présenté sur la fig. 4.1. Ainsi, la largeur totale du type paqueté *iu8vec8* est de 64 bit et nous pouvons l'utiliser pour un travail avec des architectures possédant des instructions multimédia pour ce type, notamment les processeurs compatibles SHmedia [Bra02], processeurs compatibles MMX™, les processeurs issus de l'architecture IA-64 d'Intel ou les processeurs issus de l'architecture AMD64 d'AMD.

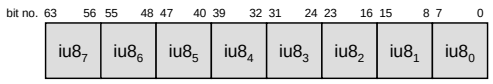


FIG. 4.1 : Structure du type de données paquetées *iu8vec8* (notation MorphoMedia [Bra05]) qui est composée de 8 éléments du type iu8 (integer unsigned 8 bit)

L'expression de ces types en formalisme fonctionnel est simple. Nous allons utiliser le type du Haskell **Array**, même si dans Haskell nous avions encore une autre possibilité - les listes. Il nous est plus naturel d'utiliser les arrays, car de leur essence, les types paquetés sont les arrays 1D. De plus, l'utilisation des arrays pour l'expression des données paquetées nous permet de faire une distinction claire entre les données statiques, exprimées par les arrays, et les données dynamiques, exprimées par les listes et décrivant les streams lors d'un traitement en flux, comme défini dans la section précédente, cf. 4.3.1.2.

Pour ces raisons, nous définissons un nouvel identificateur du type, PVec (correspondant au *pacekd vector*), en nous basant sur le type **Array** du Haskell.

```
type PVec = Array
```

Pour pouvoir initialiser une variable du type PVec, nous définissons la fonction pvec qui crée un array d'une dimension à partir d'une liste des éléments. Le premier paramètre correspond aux tuples des bornes minimales et maximales, respectivement. Le deuxième paramètre est la liste des tuples index-élément, respectivement. Cette définition est une spécialisation de la fonction standard du Haskell **array** pour une dimension.

```
pvec :: (I, I) → [(I, α)] → PVec I α
pvec i e = array i e
```

4.3.1.4 Images, arrays, vecteurs

Nous allons utiliser un type interne du Haskell, Array, pour le stockage des données matricielles et tensorielles. Ce type va nous servir au stockage d'images, textures, des arrays 2D ou des vecteurs. Pour des raisons de clarté et de simplicité de compréhension dans les définitions plus longues, nous allons définir pour ce type un nouvel identificateur plus court, Ar.

```
type Ar = Array
```

Expliquons son utilisation sur quelques exemples des signatures de type pour les array polymorphes de 1 et 2 dimensions. Elles seront employées par la suite dans les définitions des fonctions maniant ces structures de données :

- Ar I α – array d'une dimension, dont les index sont du type I et dont les élements sont du type polymorphe α,
- Ar (I, I) α – array de deux dimensions, dont les index sont des tuples (I, I) et dont les élements sont du type polymorphe α,
- Ar (I, I) **Int** – array de deux dimensions, dont les index sont des tuples (I, I) et dont les élements sont du type précis, entier, **Int**,
- Ar (I, I) (Ar (I, I) α) – array de deux dimensions, dont les index sont des tuples (I, I) et dont les éléments sont des arrays de deux dimensions Ar (I, I) α.
- Ar (I, I) (PVec I α) – array de deux dimensions, dont les index sont des tuples (I, I) et dont les éléments sont des arrays paquetés PVec I α.

4.4 Primitives du calcul comme skeletons algorithmiques

Pour pouvoir exprimer la structure des méthodes du calcul sans exposer les détails de leurs implémentations, nous allons faire appel aux fonctions du Haskell qui seront définies pour les types polymorphes (α, β, γ, etc.) Ces fonctions, entièrement génériques ou partiellement spécialisées pour des types concrets, ont le caractère des primitives du calcul et elles sont destinées à être utilisées dans les algorithmes plus complexes.

Vu qu'elles définissent formellement le principe de fonctionnement d'un algorithme, nous allons les désigner par le terme *skeletons algorithmiques*, en reprenant le terme couramment utilisé dans la littérature anglaise traitant ce sujet. Nous gardons expressément le terme anglais *skeleton* plutôt qu'utiliser ses traductions françaises possibles – *squelette*, qui peut être confondu avec les opérations morphologiques désignées par le même mot, ou *charpente* dont la forme nous semble trop éloignée du mot anglais original et où son utilisation éventuelle dans la locution *charpente algorithmique* ne nous semblerait pas appropriée.

4.4.1 Primitives du calcul séquentiel

4.4.1.1 Paradigme pipeline*, skeleton* pipe

Le skeleton pipe[DFH+93] capture la façon séquentielle et uniforme de traitement des données. Il effectue une composition chaînée des fonctions, remises dans une liste comme le paramètre d'entrée, à une fonction de sortie en utilisant la fonction foldr1 de réduction par application de la fonction de composition $\circ$. Le parallélisme est obtenu par l'allocation d'un processeur distinct pour chaque fonction. Remarquons que lors du traitement avec ce skeleton, le type polymorphe de la donnée ne change pas au cours de la progression dans le pipeline, il reste toujours α.

```
pipe  ::  [α → α] → (α → α)
pipe  =  foldr1 ( ∘ )
```

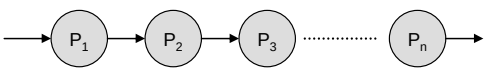


FIG. 4.2 : Skeleton algorithmique pipe

4.4.2 Primitives du calcul parallèle

4.4.2.1 Paradigme de réplication fonctionnelle*, skeleton* farm

Le skeleton farm[1] exprime la forme la plus simple du parallélisme de données. Le calcul sur ces données est perçu comme liste des tâches. Lors de la réalisation matérielle, plusieurs processeurs sont utilisés pour l'évaluation de ces tâches.

Dans notre modèle mathématique, nous procédons à l'application d'une fonction f à toutes les données d'une liste d'entrée. Dans la définition du skeleton farm, nous utilisons l'*application partielle*, cf. 4.2.1, de la fonction **map** pour laquelle nous spécifions son premier argument comme la fonction f. Ainsi, on définit une nouvelle fonction du type $([\alpha] \rightarrow [\beta])$ qui opère sur les streams, qui prend un stream d'entrée dont les éléments sont du type α et auxquels nous appliquons la fonction f. Le stream résultant de cette nouvelle fonction est du type β.

$$\begin{aligned} &\mathsf{farm} \quad :: \quad (\alpha \rightarrow \beta) \rightarrow ([\alpha] \rightarrow [\beta]) \\ &\mathsf{farm}\ f \quad = \mathbf{map}\ f \end{aligned}$$

Notons que d'un point de vue mathématique, la définition de la fonction farm est identique à celle de la fonction **map**. Ce que nous obtenons en définissant la fonction farm, c'est l'information syntaxique que nous ajoutons à la description de la fonctionnalité et qui nous indique expressément l'idée de la parallélisation du calcul. Cette information syntaxique est à exploiter lors de la compilation pour une machine parallèle.

La figure 3.9, page 44, présentée dans la section dédiée au calcul sur les streams, illustre graphiquement l'impact de cette information syntaxique sur la manière exacte d'exécuter le kernel du calcul par les architectures traitant des données en tant que flux. Nous pouvons y percevoir la différence entre le traitement d'un stream en séquence par la fonction **map** (cf. fig. 3.9(a)) et le traitement en parallèle par la réplication fonctionnelle qui est exprimée par le skeleton farm (cf. fig. 3.9(b)).

4.4.2.2 Paradigme Divide and Conquer*, skeleton* dc

La division de traitement d'une grande tâche en plusieurs plus petites qui sont traitées séparément est une approche courante sur les architectures parallèles. Les résultats partiels issus de ces petites tâches sont combinés et constituent le résultat de la tâche originale. Nous parlons d'un paradigme *divise et conquiers*[2] (de l' angl. *divide and conquer*). Bien sûr, nous pouvons l'appliquer seulement si notre problème est divisible et les parties peuvent être traitées indépendamment les unes des autres, ce qui est le cas pour la plupart des traitements des données perçues comme un flux.

Dans une architectures matérielle qui implémente ce paradigme, nous pouvons distinguer trois types de fonctionnement des unités exécutives. Ces unités peuvent être dédiées à leur fonction mais elles peuvent aussi être générales et toutes les mêmes au niveau du matériel en différant dans la fonctionnalité souhaitée qu'elles exécutent. Le premier type des unités est dédié à la division du problème, le deuxième au calcul sur les données et le troisième type est dédié à l'interprétation d'un résultat global à partir des résultats partiels. La figure 3.17, page 50, présentée dans la section consacrée au calcul stream sur les architectures SWAR à plusieurs fils d'exécution, illustre bien le principe de la division d'un stream, de l'exécution distribuée sur les streams partiels et de la collecte des résultats afin d'obtenir un seul stream résultant.

Le modèle mathématique[DFH+93] de ce paradigme est décrit par la fonction dc. Ici, les tâches triviales ($istriv$) sont solutionnées ($solv$) directement sur le processeur hôte. Les tâches plus complexes sont divisées (dvd) en plusieurs tâches plus petites qui passent dans d'autres processeurs pour y être solutionnées

[1] Nous n'utilisons pas la forme de ce skeleton décrite dans les articles[DFH+93, DGTY95b] de Darlington et al. car elle travaille avec un paramètre supplémentaire décrivant l'environnement. Ce paramètre ne figure pas dans notre définition mais peut être ajouté comme application partielle $f\$env$ de la fonction f sur l'environnement env.

[2] L'expression française correspondant au terme anglais *divide and conquer* est *diviser pour régner*. Vu que notre intérêt dans le calcul mathématique n'est pas de *régner* mais de *conquérir* les résultats à partir des unités distribuées, nous préférons utiliser la traduction libre *divise et conquiers*

($solv$) récursivement. Les résultats de ces tâches divisées sont combinés (cmb) pour obtenir le résultat global.

```
dc  ::  (α → Bool) → (α → β) → (α → [α]) → ([β] → β) → α → β
dc  istriv  solv  dvd  cmb  x
          | istriv x      = solv x
          | not(istriv x) = cmb ∘ (map $ (dc istriv solv dvd cmb)) ∘ dvd $ x
```

4.4.3 Paquetage et dépaquetage des arrays pour le traitement SIMD

Nous appelons *paquetage d'un array 1D* la transformation d'un array 1D en un autre array 1D dont les éléments sont les vecteurs paquetés (PVec). Dans la littérature on parle aussi de *vectorisation* des données (dérivée d'un terme anglais *vectorize*).

4.4.3.1 Paquetage d'un array 1D

La transformation de paquetage est triviale dans le cas d'un array 1D où l'on n'a qu'un seul axe à vectoriser et le procédé est ainsi très simple et direct. Malgré cela, nous tenons à décrire ici cette opération par l'approche fonctionnelle, car nous pouvons ainsi démontrer, sur un exemple trivial, la construction, l'indexation et le travail avec les arrays dans Haskell.

La fonction mkAr1DPVec définit cette transformation :

```
mkAr1DPVec        ::  I → Ar I α → Ar I (PVec I α)
mkAr1DPVec n ar = array bndsnew
                    [(i, pvec (1,n) [(k,ar!(lo+(i−lo)∗n+(k−1))) | k ← [1..n]]
                     ) | i ← range bndsnew]
    where
        (lo,hi) = bounds $ ar
        bndsnew= (lo, lo−1+(div (hi−lo+1) n))
```

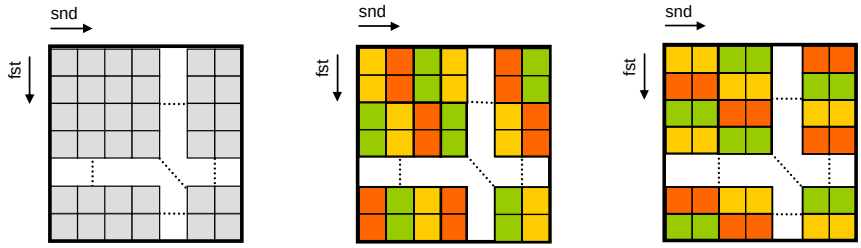
Elle prend deux paramètres. Le premier, n, définit le nombre d'éléments qui seront paquetés dans les éléments PVec de l'array de sortie. Le deuxième paramètre ar est l'array d'entrée, remarquons que sa taille doit être divisible par n. La manière de découpage de l'array d'entrée est illustré sur la fig. 4.3

FIG. 4.3 : Découpage d'un array lors de sa transformation à un array paqueté, nombre d'éléments dans un élément paqueté $n = 2$

4.4.3.2 Paquetage d'un array 2D

Le passage d'un array de 2D ou de plusieurs dimensions à un array de la même dimension composé de vecteurs paquetés PVec nécessite le choix de l'axe principal pour la vectorisation. Le choix simple est prédéfini par la manière dont les données sont stockées dans la mémoire. Sur les processeurs GPP/GPPMM, toutes les données, y compris les structures nD, sont stockées dans un espace linéaire 1D et accédées ainsi. L'accès par les instructions SIMD que nous utilisons sur les architectures GPPMM pour la lecture et la sauvegarde des données n'en fait pas exception. Si nous souhaitons utiliser ces instructions dans nos algorithmes, nous sommes contraints dans notre choix de l'axe de vectorisation de nos données par le sens de leur stockage dans la mémoire.

En revanche, si nous souhaitons paqueter les données d'un array dans l'axe perpendiculaire à celui de la mémoire, notre travail exige une autre approche car nous ne pouvons pas accéder à ces données directement par les instructions SIMD. Nous sommes obligés d'utiliser soit la lecture élément par élément, soit une autre approche qui nous permettrait d'interpréter correctement les données lues dans un



(a) array 2D avant découpage

(b) découpage par la direction de la première coordonnée (fst)

(c) découpage par la direction de la deuxième coordonnée (snd)

FIG. 4.4 : Exemple de vectorisation d'un array 2D pour différentes versions de découpage et la taille du vecteur paqueté $n = 2$

sens différent. Nous montrerons les algorithmes implémentant cette approche dans le chapitre 6 dédié à la permutation des arrays, page 127.

Nous allons présenter deux possibilités de vectorisation d'un array 2D : le paquetage par l'axe de la première coordonnée défini par la fonction mkAr2DPVecByFst et le paquetage dual par l'axe de la deuxième coordonnée par la fonction mkAr2DPVecBySnd. Pour présenter une définition générique et indépendante d'un système de coordonnées dans l'image, nous ne parlons pas des coordonnées x ou y d'une image mais nous utilisons la notation matricielle et parlons des coordonnées première et deuxième, exprimées par les suffixes Fst ou Snd respectivement.

D'un point de vue pratique, ces définitions ne correspondent pas à un algorithme concret. Il s'agit, en effet, de la formalisation du changement de la perception des données stockées dans la mémoire et du changement de leur mode d'adressage. On peut dire que ces définitions correspondent au changement du type des données à partir d'un array du type Ar (I, I) α à un array du type Ar (I, I) (PVec I α) dont les éléments sont les vecteurs paquetés, exactement comme c'est inscrit dans la signature de type de ces fonctions.

La fonction mkAr2DPVecByFst définit la vectorisation par l'axe de la première coordonnée et la manière dont l'array est découpé en données paquetées est illustrée sur la fig. 4.4(b)

```
mkAr2DPVecByFst          :: I → Ar (I,I) α → Ar (I,I) (PVec I α)
mkAr2DPVecByFst n ar = array bndsnew
                       [((i,j), pvec (1,n) [(k,ar!(flo+(i−flo)∗n+(k−1),j)) | k ← [1..n]])
                        | (i,j) ← range bndsnew]
    where
        ((flo,slo),(fhi,shi)) = bounds $ ar
        bndsnew= ((flo,slo),(flo−1+(div (fhi−flo+1) n), shi))
```

Cette fonction crée un nouvel array par la fonction **array**. Cet array a une dimension réduite dans l'axe de paquetage. Les éléments de cet array sont les vecteurs paquetés PVec, créés par la fonction pvec. La fonction **bounds** est utilisée pour obtenir les limites de l'array d'entrée *ar*. La variable *bndsnew* détient les nouvelles limites de l'array de sortie.

La fonction similaire, mkAr2DPVecBySnd, définit la vectorisation par l'axe de la deuxième coordonnée et la manière de découper est illustrée sur la fig. 4.4(c)

```
mkAr2DPVecBySnd          :: I → Ar (I,I) α → Ar (I,I) (PVec I α)
mkAr2DPVecBySnd n ar = array bndsnew
                       [((i,j), pvec (1,n) [(k,ar!(i,slo+(j−slo)∗n+(k−1))) | k ← [1..n]])
                        | (i,j) ← range bndsnew]
    where
        ((flo,slo),(fhi,shi)) = bounds $ ar
        bndsnew= ((flo,slo),(fhi, slo−1+(div (shi−slo+1) n)))
```

Nous définissons également une fonction commune, mkAr2DPVec qui prend un paramètre de plus qui nous sert comme clé dans le choix de soit la fonction mkAr2DPVecByFst ou mkAr2DPVecBySnd :

```
mkAr2DPVec    ::  [Char] → I → Ar (I,I) α → Ar (I,I) (PVec I α)
mkAr2DPVec how n ar | how == "Fst" = mkAr2DPVecByFst  n ar
                    | how == "Snd" = mkAr2DPVecBySnd  n ar
```

4.4.3.3 Dépaquetage des arrays

Nous appelons le *dépaquetage* d'un array le processus d'obtention d'un array composé des éléments de base à partir d'un array dont les éléments sont des vecteurs paquetés. Il est possible de définir les fonctions de dépaquetage qui auront des fonctionnalités inverses à celles que l'on vient de définir pour le paquetage. Leurs définitions sont intelligibles et nous ne présentons que leurs identificateurs et les signatures de type.

La fonction mkAr1DFromAr1DPVec définit la fonction de dépaquetage pour un array 1D et sa signature de type est la suivante :

```
mkAr1DFromAr1DPVec:: Ar I (PVec I α) → Ar I α
```

Les fonctions mkAr2DFromAr2DPVecByFst et mkAr2DFromAr2DPVecBySnd définissent le dépaquetage d'un array 2D dans le sens de la première/deuxième coordonnée, respectivement. La fonction mkAr2DFromAr2DPVec les englobe dans une seule qui choisit entre la première/deuxième fonction selon la valeur correspondante *"Fst"* ou *"Snd"* de son premier paramètre textuel (donnée par [Char]). Voici leurs signatures de type :

```
mkAr2DFromAr2DPVecByFst  ::  Ar (I,I) (PVec I α) → Ar (I,I) α
mkAr2DFromAr2DPVecBySnd  ::  Ar (I,I) (PVec I α) → Ar (I,I) α
mkAr2DFromAr2DPVec       ::  [Char] → Ar (I,I) (PVec I α) → Ar (I,I) α
```

4.4.4 Sens du parcours, passage d'un array à un flux de données et vice versa

4.4.4.1 Notion du sens du parcours et de l'extraction des éléments

Le passage d'une structure statique de données tel qu'un vecteur ou array 2D à une structure utilisée dynamiquement pour le traitement en flux nécessite le choix du sens de parcours.

Pour nos besoins, nous allons comprendre sous le terme *parcours* d'un array une séquence des index qui désignent les éléments d'un array. Nous parlons ainsi d'un stream des index et ce stream peut, généralement, ne désigner qu'un sous-ensemble des éléments de cet array. Si les index de ce stream désignent tous les éléments d'un array, nous parlons d'un *parcours complet* d'un array.

Une fois le parcours défini, le passage d'un array à un flux de données sera obtenu par l'application de l'opération indexation des éléments, exprimée par la fonction ! du Haskell, sur le stream des index. À cette occasion, nous allons parler d'*extraction des éléments* ou également d'*échantillonnage* d'un array.

Ainsi perçu, l'accès à la mémoire qui est nécessaire pour l'obtention des éléments d'un array est exécuté en tant qu'opération sur le stream et la fonction ! d'échantillonnage de l'image devient, en effet, le kernel du calcul sur les flux de données. Nous pouvons le décrire dans le formalisme fonctionnel par l'expression suivante :

```
(map (ar!)) $ (strm ar)
```

où *ar* est l'array d'entrée, *strm* désigne la fonction de parcours qui crée un stream des index à partir de l'array *ar*. Elle est suivie par l'application de la fonction **map** sur ce stream qui se charge d'exécuter l'indexation de l'array *ar*! sur chacun des index de ce stream. La figure 4.5 illustre cette situation. Le parcours de l'image y est représenté par le bloc de la fonction génératrice des index qui est succédé par le bloc d'extraction des éléments de la mémoire.

Cette modélisation mathématique que nous introduisons ici et qui perçoit l'accès à la mémoire comme opération sur les streams est très intéressante d'un point de vue pratique. En tant qu'opération

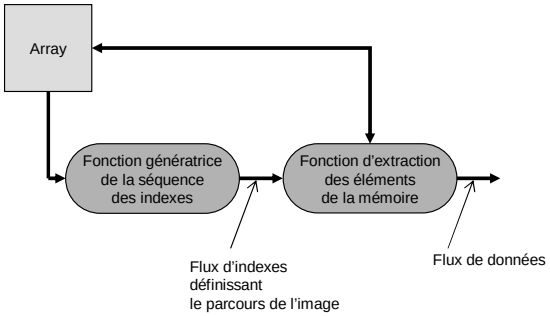


FIG. 4.5 : Passage d'un array à un flux de données est effectué dans la logique des kernels d'exécution

sur les streams, elle peut profiter de toutes les techniques de parallélisation de traitement des streams, notamment de celle de la réplication fonctionnelle modélisée par le skeleton farm, cf. page 67. L'accès concurrent à la mémoire est difficile à imaginer sur les architectures classiques de Von Neumann, mais c'est une technique connue et utilisée sur les machines parallèles ou des architectures dédiées.

C'est cette approche de *parcours* de l'array et d'*extraction* des éléments que nous allons utiliser pour le passage d'un array à un flux de données. Cependant, les fonctions de parcours de l'image vont nous servir également lors de la recomposition d'un array de sortie à partir d'un flux de données. Il s'agit, en effet, du processus inverse au passage à flux de données et pour l'exprimer en formalisme fonctionnel, nous allons utiliser la fonction standard **array** du Haskell de création d'un array.

La fonction **array** prend deux arguments. Le premier argument est un tuple des bornes minimales et maximales des index. Nous reconstituons un array de sortie à partir d'un array d'entrée qui a les mêmes bornes maximales et minimales. Nous pouvons utiliser directement la fonction du Haskell qui fournit ces informations, **bounds** avec l'array d'entrée comme paramètre. Le deuxième argument de la fonction **array** est une liste des tuples (index, valeur) qui doit contenir tous les éléments inclus dans les bornes. Nous construisons cette liste à partir de notre stream des résultat et à partir du stream des index, le même que nous avons utilisé pour parcourir l'array. Nous les associons élément par élément avec la fonction **zip** du Haskell.

Voici un exemple de cette construction :

```
array (bounds $ ar) (zip ixs (id ss))
    where ixs = strm ar; ss = map (ar!) ixs
```

où *ar* désigne l'array d'entrée, *ixs* est le stream des index créé par la fonction *strm* du parcours de l'array et *ss* est le stream des valeurs des résultats. La fonction d'identité **id** nous indique l'endroit où nous plaçons des fonctions exécutives effectuant un traitement sur le stream des valeurs.

4.4.4.2 *Fonction* indices*, fonction standard du Haskell pour le parcours d'un array*

Dans le cas où nous avons besoin de parcourir l'array entier mais sans poser de contrainte sur le sens du parcours, nous pouvons utiliser la fonction standard **indices** du Haskell. Elle nous retourne des indices de tous les éléments de l'array par l'indexage de ses bornes.

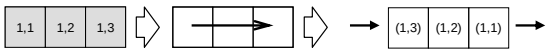
Sachant que nous avons utilisé une fonction standard pour le passage à un flux de données, nous pouvons utiliser un mécanisme plus simple pour la recomposition de l'array de sortie en utilisant la fonction **listArray** du Haskell. Cette dernière n'exige pas l'association des éléments du flux des résultat avec l'index mais travaille directement avec ce stream. L'exemple suivant illustre cette situation :

```
listArray (bounds $ ar) (id ss)
    where ixs = indices ar; ss = map (ar!) ixs
```

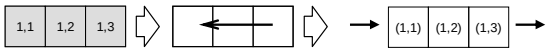
4.4.4.3 Fonctions de parcours d'un array

Commençons notre explication par l'introduction de la fonction concrète de parcours d'un array 1D. La fonction streamAr1D définit un skeleton algorithmique qui crée un stream des index à partir d'un array 1D. La valeur passée par son paramètre *how* détermine le sens du parcours que nous voulons obtenir, la valeur "FW" correspond au sens *au-devant*, la valeur "BW" correspond au sens *en arrière*. La fig. 4.6 illustre ces deux cas sur un vecteur de la taille 3.

```
streamAr1D  ::  [Char] → Ar  I  α → [I]
streamAr1D how ar
    | how == "FW" = [(lo+i)  |  i ← [0..size−1]]
    | how == "BW"= [(hi−i)  |  i ← [0..size−1]]
  where
    (lo,hi)  = bounds $ ar; size = rangeSize(lo,hi)
```



(a) sens au-devant, streamAr1D avec "FW"



(b) sens en arrière, streamAr1D avec "BW"

FIG. 4.6 : Choix du parcours de l'image pour un array de 1D

Tandis que pour les structures 1D le choix du sens de parcours est simple et nous utilisons soit le parcours au-devant, soit le parcours en arrière, pour les structures 2D nous avons beaucoup plus de possibilités et nous décrivons celles qui sont utilisées le plus souvent. Il n'est pas difficile, en cas de besoin, d'en définir davantage qui seraient appropriées à un traitement particulier.

Mais tout d'abord, nous définissons un type Streamize que nous allons utiliser pour désigner une fonction de parcours d'un array de 2D dans les signatures de types de nos algorithmes :

```
type Streamize α =  Ar  (I,I)  α → [(I,I)]
```

Il s'agit des fonctions qui prennent un array 2D comme argument et qui nous retournent un stream des index.

Nous définissons un skeleton pour la création d'un stream des index à partir d'un array 2D par la fonction streamAr2D qui définit les parcours bien connus d'une image en sens *vidéo* et en sens *anti-vidéo*.

```
streamAr2D  ::  [Char] → Ar  (I,I)  α → [(I,I)]
streamAr2D how ar
    | how == "FWFst" = [(flo+i,slo+j)  |  j←[0..smax], i←[0..fmax]]
    | how == "FWSnd" = [(flo+i,slo+j)  |  i←[0..fmax], j←[0..smax]]
    | how == "BWFst" = [(fhi−i,shi−j)  |  j←[0..smax], i←[0..fmax]]
    | how == "BWSnd" = [(fhi−i,shi−j)  |  i←[0..fmax], j←[0..smax]]
  where
    ((flo,slo),(fhi,shi))  = bounds $ ar;
    fmax = rangeSize(flo,fhi)−1
    smax = rangeSize(slo,shi)−1
```

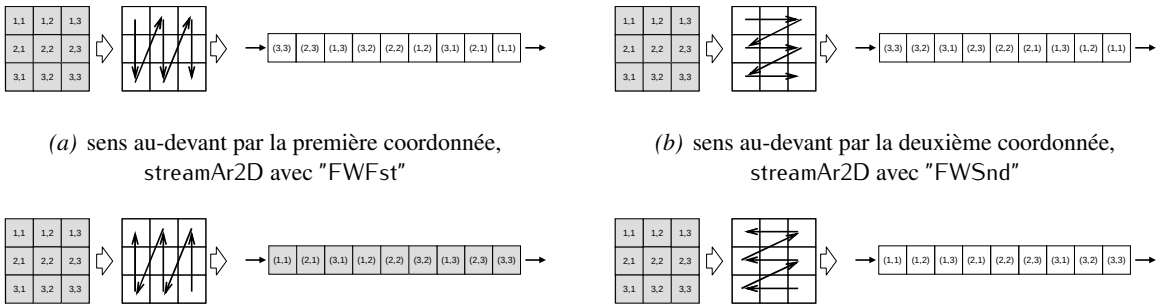
Le type du parcours exact est déterminé par la valeur du paramètre *how* de cette fonction et correspond pour les valeurs "FWFst" / "FWSnd" aux sens *au-devant par la première / deuxième coordonnée*, et pour les valeurs et "BWFst" / "BWSnd" aux sens *en arrière par la première / deuxième coordonnée*, respectivement.

Notons que l'application partielle de cette fonction avec la paramètre *how* concret, e.g. :

```
(streamAr2D$"FWFst")
```

nous définit dans Haskell une nouvelle fonction qui est du type Streamize α et qui est ainsi directement utilisable dans les algorithmes comme une fonction du parcours d'un array.

La figure 4.7 illustre le fonctionnement de ce skeleton pour un array 2D 3x3.



(a) sens au-devant par la première coordonnée, streamAr2D avec "FWFst"

(b) sens au-devant par la deuxième coordonnée, streamAr2D avec "FWSnd"

(c) sens en arrière par la première coordonnée, streamAr2D avec "BWFst"

(d) sens en arrière par la deuxième coordonnée, streamAr2D avec "BWSnd"

FIG. 4.7 : Choix du parcours de l'image pour un array de 2D 3×3

4.4.5 Concept des "superpixels"

Dans certains de nos algorithmes, nous allons travailler avec un groupe de pixels et avec les pixels voisins de ce groupe, plutôt que de travailler avec un seul pixel et son voisinage. Il serait donc convenable de décrire à cette place la manière dont on va travailler avec un tel groupe et de donner les bases formelles à ce travail.

L'idée de travailler avec des groupes de pixels et des pixels voisins de ce groupe est propre à toutes les implémentations sur les architectures parallèles où on procède à la division de l'image et on effectue le traitement de ces parties par la distribution sur différents processeurs. Cette idée est explorée par Jin Yang qui utilise, q.v. page 57 de sa thèse [Yan97] doctorale, les bords partagés (également appelés les *halos*) qui sont ajoutés aux arrays parallèles ou aux graphes. Notons que ces bords partagés ont dans la morphologie mathématique un sens plus large que celui du voisinage proche sur une grille donnée car ils doivent refléter le travail avec des éléments structurants dans leur forme générale et souvent d'une taille importante, pas nécessairement celle qui définit le voisinage de taille 1.

D'un point de vue formel, c'est le travail sur le voisinage qui nous empêche d'exprimer un tel groupe de pixels par un array découpé régulièrement sur les vecteurs paquetés ou sur les macro blocs car dans la perception des arrays comme nous la décrivons par le formalisme fonctionnel, les voisins d'une donnée qui est du type α sont les données du même type. Par exemple, le voisinage d'un macro bloc concret est constitué d'un ou plusieurs macro blocs voisins.

Ceci ne correspond pas à la philosophie de travail que nous voulons employer pour les groupes de pixels dont le voisinage (proche ou dans le sens large) est constitué également des pixels et non des groupes de pixels. Vu que l'utilisation de ces groupes de pixels dans les fonctions et la façon de travailler avec leurs pixels voisins sont semblables à celles que nous employons lors du travail à l'échelle des pixels non-groupés, il nous semble approprié de désigner ces groupes comme des *superpixels*, c'est-à-dire des pixels qui ont une notion élargie d'une entité de données qui peut contenir plus d'un seul pixel.

Le concept des superpixels que nous introduisons est pratique pour deux raisons :

- il est cohérent avec l'idée du sens du parcours implémentée par la fonction génératrice des index et avec l'idée d'une fonction d'extraction des pixels (ici de tout un groupe) à partir de l'image en utilisant un seul index, comme nous l'avons présenté dans la section 4.4.4, page 70.
- les superpixels conservent le caractère des pixels. Ainsi, nous n'avons pas besoin de percevoir les groupes de pixels très différemment (e.g. comme des macro blocs), de définir un autre type qui serait spécifique aux groupes de pixels et qui nous aurait conduit à un travail très différent de

celui pour les pixels. Un travail très différent surtout parce que nous aurions besoin d'introduire des fonctions plus élaborées d'extraction des pixels voisins (ordinaires) à partir d'un groupe qui serait de ce nouveau type. C'est pourquoi nous préférons introduire des superpixels pour lesquels nous pouvons utiliser, lors de la construction de nos algorithmes, la même charpente que celle utilisée pour les pixels ordinaires ; même si, bien sûr, nous aurons besoin de modifier certains points spécifiques.

Remarquons que l'idée des superpixels n'est pas restreinte uniquement aux arrays définis sur les grilles régulières mais peut être transposée au traitement général des graphes. Pour ces derniers, un *superpixel* serait défini comme un groupe des sommets du graphe qui peuvent être traités en même temps.

4.4.5.1 Travail avec des superpixels

La position d'un superpixel dans l'array est définie par un index que nous allons appeler l'*index d'ancrage*. Sa valeur doit être incluse dans les bornes minimales et maximales de cet array. De plus, l'élément de l'array qui est désigné par l'index d'ancrage d'un superpixel doit appartenir au groupe des éléments constituant ce superpixel.

La fonction d'échantillonnage des superpixels décrit la manière dont les éléments d'un superpixel sont extraits à partir de l'image. Étant donné un array ar et l'index d'ancrage i d'un superpixel, les éléments de ce dernier peuvent être obtenus par l'application d'une fonction concrète $sampFncSP$ d'extraction de superpixels :

```
sampFncSP ar i
```

L'ensemble de tous les superpixels d'un array doit obligatoirement composer l'array entier. Le nombre précis des éléments composant un superpixel peut être variable d'un superpixel à l'autre. Deux cas spéciaux peuvent être distingués, celui d'un superpixel composé d'un seul élément et celui d'un superpixel composé de tous les éléments d'un array. La définition exacte des superpixels n'est pas restreinte par d'autres conditions, on n'exige pas une forme géométrique particulière ni que cette forme soit convexe.

4.4.5.2 Sens du parcours, passage d'un array à un flux de superpixels et vice versa

Cependant, pour le travail pratique, il est préférable de définir des superpixels d'une manière unifiée comme des ensembles d'éléments définissant le pavage de l'array aux zones rectangulaires de mêmes dimensions dans lesquelles nous choisissons par convention les index les plus petits comme les index d'ancrage. La figure 4.8 illustre cette situation.

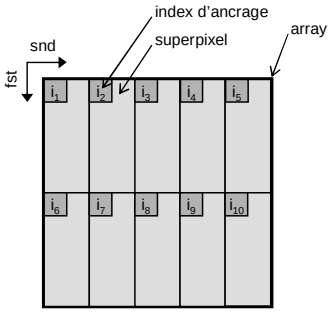


FIG. 4.8 : Décomposition d'un array aux superpixels rectangulaires de mêmes dimensions

Dans ce but, nous définissons la fonction streamAr2DSP du sens du parcours pour les superpixels qui nous retourne un stream des index d'ancrage des superpixels et dont le fonctionnement est très semblable à celui de la fonction streamAr2D, page 72 :

```
streamAr2DSP    :: [Char] → I → I → Ar (I,I) α → [(I,I)]
streamAr2DSP  how m n ar
    | how == "FWFst" = [(flo+m∗i, slo+n∗j)  | j←[0..smax], i←[0..fmax]]
    | how == "FWSnd" = [(flo+m∗i, slo+n∗j)  | i←[0..fmax], j←[0..smax]]
    | how == "BWFst" = [(fhi−m∗i, shi−n∗j)  | j←[0..smax], i←[0..fmax]]
    | how == "BWSnd" = [(fhi−m∗i, shi−n∗j)  | i←[0..fmax], j←[0..smax]]
  where
    ((flo, slo), (fhi, shi))  = bounds $ ar;
    fmax = div (rangeSize(flo,fhi)−1) m
    smax = div (rangeSize(slo,shi)−1) n
```

Le premier argument est la clé avec laquelle nous désignons la fonctionnalité exacte de cette fonction. Le deuxième / troisième argument de cette fonction désigne les dimensions d'un superpixel dans la première/deuxième coordonnée. Le quatrième paramètre est l'array d'entrée que nous voulons parcourir.

Nous définissons le type StreamizeSP qui va désigner des fonctions pour le passage d'un array à un stream des index d'ancrage des superpixels. La signature de ce type est, en effet, identique aux fonctions désignées par le type Streamize. Tandis que les fonction étant du type Streamize travaillent avec les index ordinaires, le type StreamizeSP porte, de plus, une information syntactique du type de retour $[(I, I)]$, car nous le définissons comme la liste des index d'ancrage des superpixels :

```
type StreamizeSP α = Ar (I,I) α → [(I,I)]
```

Ainsi, la signature de type de la fonction streamAr2DSP que nous venons de présenter :

```
streamAr2DSP :: [Char] → I → I → Ar (I,I) α → [(I,I)]
```

peut être récrite comme :

```
streamAr2DSP :: [Char] → I → I → StreamizeSPα
```

Ayant obtenu un stream des index d'ancrage par la fonction streamAr2DSP, nous avons encore besoin des fonctions qui extrairaient à partir de ce stream les éléments de bases de l'image qui composent les superpixels correspondants. Ces fonction seront du type SampFncSP :

```
SampFncSPα :: (Ix β) ⇒ Ar β α → β → [α]
```

et elle diffèrent des versions classiques (non-superpixeliques) des fonctions d'échantillonnage dans le type de retours qui est dans ce cas une liste des éléments.

Ce procédé est assuré, dans la version générale, par la fonction sampSPGen. Elle va nous retourner, pour un index d'ancrage donné, la liste de tous les éléments de l'array appartenant au superpixel qui est désigné par cet index.

```
sampSPGen     :: I → I → Ar (I,I) α → (I,I) → [α]
sampSPGen m n ar (ixf,ixs) = map (ar!) ( range((ixf,ixs), (ixf+m−1, ixs+n−1)) )
```

Notons que la signature de type de cette fonction est compatible avec celle qui utilise le type SampFncSP :

```
sampSPGen     :: I → I → SampFncSPα
```

et nous pouvons l'utiliser, après l'application partielle de ses deux premiers paramètres, comme la fonction d'entrée dans les algorithmes exigeants les fonction du type SampFncSP.

Nous utiliserons encore une autre catégorie de fonctions qui est connexe aux superpixels et qui sera utilisée lors de la recomposition de l'array de sortie. Il s'agit des fonctions qui, à partir d'un tuple composé de l'index d'ancrage et de la liste des éléments résultants d'un superpixel, créent la liste des tuples (index, élément). Cette dernière liste sera utilisée dans les algorithmes comme un moyen pour pouvoir recomposer le stream d'entrée de la fonction standard *array* de Haskell afin de créer l'array de sortie. Ces fonctions seront du type ZipSP :

```
type ZipSP α = ((I,I),[α]) → [((I,I), α)]
```

La fonction qui sera de cette catégorie, qui complète le passage au flux de superpixels, comme défini par la fonction StreamAr2DSP et leur extraction, comme défini par la fonction sampSPGen, c'est la fonction zipSPGen

```
zipSPGen     :: I → I → ((I,I),[α]) → [((I,I), α)]
zipSPGen m n ((ixf,ixs), ss) = zip ( range((ixf,ixs), (ixf+m−1, ixs+n−1)) ) ss
```

Elle prend deux paramètres supplémentaires, *m* et *n* définissant les dimensions des superpixels dans l'axe de la première et la deuxième coordonnée, respectivement. En effet, la signature de type de cette fonction est compatible avec la suivante :

```
zipSPGen :: I → I → ZipSP
```

qui utilise le type ZipSP et démontre plus explicitement sa désignation.

Regardons maintenant comment nous allons utiliser les fonctions de travail avec les superpixels sur un exemple trivial. Dans cet exemple, en dehors du passage d'un array *ar* à un stream des superpixels de dimensions $m \times n$ et de la recomposition d'un nouvel array à partir de ces derniers, nous n'employons aucune fonction de traitement des superpixels.

```
array (bounds ar)
(
   (foldl1 (++))
 ∘ (map (zipSPGen m n))
 ∘ (zip ixs)
 ∘ (map (sampSPGen m n ar))
 $ ixs
)
where
   ixs = streamAr2DSP "FWFst" m n ar
```

Tout d'abord, nous choisissons le parcours de l'image par la fonction streamAr2DSP et nous obtenons un stream des index d'ancrage. Sur ce stream, nous appliquons la fonction d'extraction des superpixels sampSPGen pour obtenir le stream des superpixels qui est du type liste des listes, $[[\alpha]]$. Lors de la recomposition, nous ajoutons à chacun des superpixels son index d'ancrage par la fonction **zip** pour obtenir un stream des tuples (index d'ancrage, superpixel). Sur tous les éléments de ce stream, nous appliquons, par la fonction **map**, la fonction zipSP retournant une liste des tuples (index, élément). Nous connectons ces listes distinctes par l'application de la fonction **foldl1** de réduction d'un stream par la fonction ++ de jonction des listes. La figure 4.9 illustre graphiquement le fonctionnement de ce procédé.

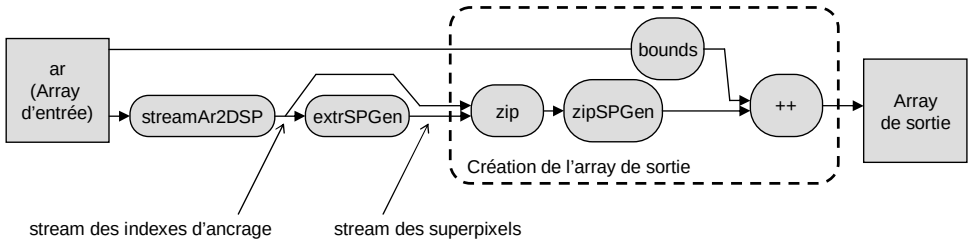


FIG. 4.9 : Exemple de travail avec les streams des superpixels

4.5 Modèle formel du traitement en pipeline graphique

4.5.1 Types de données utilisés dans le modèle

Nous allons utiliser la même logique que celle suivie pour la définition des types de base, cf. 4.3.1, également pour les définitions de nouveaux types pour le modèle du traitement en pipeline graphique sur les GPU.

Ces définitions que nous nous apprêtons à décrire devraient être perçues par le lecteur comme des définitions sémantiques, les nouveaux identificateurs nous permettront de distinguer le sens des paramètres dans les fonctions plus complexes que nous allons rencontrer.

Il existe plusieurs types de données qui sont utilisés par les GPU modernes pour le traitement et peuvent varier le long du calcul, s'agissant de nombres entiers ou de nombres à une virgule flottante à diverses précisions. Ainsi, il est possible d'avoir les données stockées dans les textures en nombres entiers mais d'effectuer le calcul dans le pipeline graphique en virgule flottante.

Pour pouvoir rester facilement compréhensible dans nos prochaines explications, nous n'allons pas chercher à construire un modèle le plus complet possible pour le GPU, même si cela pourrait être envisagé. Pour le stockage de données sur le GPU, nous allons utiliser un type de nombres entiers **Int** du Haskell. Nous appliquons cette approche sans perte d'utilité puisque les algorithmes que nous décrivons emploient des nombres entiers pour l'expression des pixels dans les images. C'est également le cas pour les opérations morphologiques dans le domaine digital. Les types relatives à l'indexation seront des types de nombres entiers, basés sur le type d'indexation de base I que l'on a déjà défini dans 4.3.1.1.

Un autre point est à remarquer. Même si les possibilités des GPU actuels permettent aussi le travail avec les données 3D en utilisant les textures 3D, nous ne travaillerons qu'avec les algorithmes travaillant avec les images 2D. Nos données pourrons ainsi contenir plusieurs composantes par pixel mais elles resteront de dimension 2. Cela va nous conduire à des définitions spécialisées pour 2 dimensions et surtout à une indexation par tuple (I, I).

Pour donner une vue globale des différents types que nous expliquons par la suite, nous présentons la table 4.1 qui récapitule leurs identificateurs, désignation et définition.

Type	Désignation	Définition
CElmnt	Élément de couleur	**type** CElmnt = **Int**
C	Vecteur de couleur	**type** C = PVec I CElmnt
CI	Index de couleur	**type** CI = I
Pos	Coordonnée de position	**type** Pos = I
P	Vecteur de position	**type** P = PVec I Pos
Dpth	Profondeur	**type** Dpth = I
TXB	Bord de la texture	**type** TXB = C
TX	Texture	**type** TX = (Ar (I,I) C, [TXB])
TXP	Position dans la texture	**type** TXP = (I,I)
TXI	Index de texture	**type** TXI = I
Shape	Primitive de la géométrie	**data** Shape = Rect I Line I Point
V	Vertex	**type** V = (P, [(CI,C)], [(TXI, TXP)])
F	Fragment	**type** F = (P, Dpth, [(CI,C)], [(TXI, TXP)])
FBO	Frame buffer object	**type** FBO *tfbo* = Ar (I,I) *tfbo*
FB	Frame buffer	**type** FB = (Ar I (FBO C), [FBO Dpth])
PX	Pixel	**type** PX = (P, Ar I C)
Env	Environnement	**type** Env = (FB, [TX])
Commands	Commandes graphiques	**type** Commands = ([Shape],[V])

TAB. 4.1 : Types de données pour les algorithmes utilisant le pipeline graphique et les GPU

4.5.1.1 Types pour les couleurs

Nous définissons deux types pour la manipulation de la couleur sur les GPU. Le type CElmnt représente une seule composante de la couleur et sera défini pour les besoins de cette thèse en utilisant le type de nombres entiers **Int**.

```
type CElmnt = Int
```

Le deuxième type que nous définissons ici, type C, représente la couleur de plusieurs composantes que nous percevons comme un vecteur. Dans notre approche, ce vecteur a son correspondant dans le

type PVec, cf. 4.3.1.3, page 64. Même si l'on pouvait choisir le type plus général d'un array Ar et le spécialiser pour une dimension, nous utilisons ici le type de données paquetées PVec car c'est ainsi que la plupart des GPU perçoivent les vecteurs de couleurs et peuvent les traiter à l'aide des instructions SIMD à l'intérieur des registres. Ainsi nous définissons le type C comme :

```
type C = PVec I CElmnt
```

Pour pouvoir indexer les couleurs à l'intérieur d'un vecteur de couleur C, nous définissons le type CI. Il représente l'index de couleur et est basé sur le type d'indexation de base I, cf. 4.3.1.1 page 64 :

```
type CI = I
```

Nous utiliserons également deux fonctions de manipulation, c3e et c4e, qui créent un vecteur à partir des éléments et dont les définitions exactes sont présentées en Annexe B, page 201.

4.5.1.2 Types pour les coordonnées

Nous définissons les types pour la manipulation avec les coordonnées. Le type Pos est le type de base pour exprimer la position et il s'appuiera sur le type d'indexation de base I, cf. 4.3.1.1 page 64 :

```
type Pos = I
```

Le type P est un type composé et il sera utilisé pour désigner un point dans l'espace d'une ou plusieurs dimensions. Nous le définissons en utilisant le type de vecteur paqueté PVec, cf. 4.3.1.3, page 64, car nous supposons que la plupart des GPU utilisent les types et le traitement SIMD des coordonnées.

```
type P = PVec I Pos
```

Nous définissons également deux fonctions de manipulation, p2D et p3D, qui créent à partir d'un tuple ou triple de coordonnées un vecteur des coordonnées 2D ou 3D respectivement et dont les définitions exactes sont présentées en Annexe B, page 201.

Pour exprimer la profondeur dans le traitement des fragments, nous définissons le type Dpth (une abréviation du terme anglais *Depth*). Puisqu'il s'agit d'un type d'indexage, nous nous basons sur le type I, cf. 4.3.1.1, page 64 :

```
type Dpth = I
```

4.5.1.3 Types pour les textures

Les textures sont les objets qui stockent sur les cartes graphiques les images de travail. Nous définissons tout d'abord le type TXB qui exprime la valeur de bord d'une texture et qui est compatible avec le type vecteur de couleur C :

```
type TXB = C
```

La texture est définie par le type TX en tant que tuple où le premier élément est un array qui contient les données de couleur. Il est utilisé pour stocker les images 2D. Le deuxième élément est une liste qui peut être soit vide, dans le cas où nous ne travaillons pas avec les bords, soit contenir l'information sur la couleur de bord exprimé par le type TXB :

```
type TX = (Ar (I,I) C, [TXB])
```

Nous utiliserons également trois fonctions de manipulation. La fonction mkTX qui crée une texture à partir de ses composantes, les fonctions getArFromTX et getTXBFromTX nous aident à obtenir les composantes à partir d'une texture. Leurs définitions exactes sont présentées en Annexe B, page 201.

Pour pouvoir indexer un élément dans une texture, nous allons utiliser les échantillonneurs (samplers) qui travaillent avec les positions dans la texture pour ensuite en extraire une valeur. Le type TXP va exprimer la position dans la texture 2D et est défini comme :

```
type TXP = (I,I)
```

Si nous travaillons avec plusieurs textures à la fois stockées dans un array, nous aurons besoin d'un index pour y accéder. Nous définissons ainsi un type TXI destiné à ce travail en nous basant sur le type d'indexation I, cf. 4.3.1.1, page 64 :

```
type TXI = I
```

4.5.1.4 Type pour les primitives de la forme

Nous allons travailler avec certaines formes géométriques pour passer les commandes graphiques dans les unités de calcul d'un GPU. Il s'agit précisément d'un rectangle (Rect), d'une ligne (Line) et d'un point (Point). Pour les stocker, nous avons défini un type énumératif Shape :

```
data Shape = Rect | Line | Point
```

4.5.1.5 Types pour les vertex

Les vertex constituent un des piliers de traitement sur les GPU. Ce sont les structures de données utilisées pour stocker les informations relatives au traitement des vertex dans le pipeline graphique.

Formellement, nous définissons un vertex comme une structure composée, qui doit contenir obligatoirement un point P exprimant la position dans l'espace 2D ou 3D. Il peut contenir une éventuelle information sur la couleur représentée, soit par la liste vide dans le cas où le vertex ne possède pas d'information de couleur, soit par la liste des tuples (CI,C). L'index CI, est utilisé pour distinguer les données lors du travail avec plusieurs couleurs mais aussi pour mettre la couleur en correspondance avec les plans de rendu si nous utilisons le pipeline graphique pour le rendu dans plusieurs textures (angl. *render to multiple textures*). Un vertex peut contenir également aucune, une ou plusieurs information sur les données stockées dans les textures exprimées par la liste des tuples (TXI, TXP). L'index TXI identifie la texture, la position TXP identifie la position d'un élément dans cette texture.

Dans le formalisme fonctionnel, un vertex V est défini comme :

```
type V = (P, [(CI,C)], [(TXI, TXP)])
```

Nous utiliserons également une fonction de manipulation mkV qui crée un vertex à partir de ses composantes et dont la définition exacte est présenté en Annexe B, page 202.

4.5.1.6 Type pour les fragments

Les fragments constituent également un pilier de traitement sur les GPU. Ce sont les structures de données qui stockent l'information relative au traitement des fragments dans le pipeline graphique.

Un fragment est une donnée composée qui contient obligatoirement un point P de position 2D à l'écran et la profondeur du fragment Dpth. En effet, il contient encore l'information 3D à travers la profondeur. À ces données nous ajoutons, de la même façon que l'on a fait pour les vertex, des informations sur la couleur en utilisant la liste des tuples (CI,C) et des informations sur les données stockées dans les textures en utilisant la liste des tuples (TXI, TXP).

Ainsi, nous définissons le type F d'un fragment comme :

```
type F = (P, Dpth, [(CI,C)], [(TXI, TXP)])
```

Nous utiliserons aussi une fonction de manipulation mkF qui crée un fragment à partir de ses composantes et dont la définition exacte est présentée en Annexe B, page 202.

4.5.1.7 Types pour le framebuffer

Le framebuffer est une structure dans laquelle le pipeline graphique écrit les pixels à la fin du traitement, nous pouvons dire que c'est une mémoire de sortie. De nos jours, les possibilités de programmation

des GPU nous permettent de connecter plusieurs objets (plusieurs types de zones de mémoire) à la sortie du pipeline graphique et d'avoir ainsi plusieurs possibilités de stockage des résultats.

Les objets qui peuvent être connectés à un framebuffer sont appelés *Framebuffer objects* et nous définissons un type FBO pour ce but. Il s'agit d'un array de 2D dont les éléments sont du type tfbo.

```
type FBO tfbo = Ar (I,I) tfbo
```

Le type tfbo est un paramètre dans cette définition et il nous permettra de distinguer les instances des objets du framebuffer qui diffèrent dans les types d'éléments. Parmi d'autres, nous allons utiliser les FBO aussi pour le rendu dans les textures, dont la définition est très proche et diffère seulement dans la spécialisation du type pour les couleurs, cf. 4.5.1.3, page 78.

Le *Framebuffer*, qu'il ne faut pas confondre avec le *framebuffer object*, doit contenir au moins un objet FBO correspondant à la couleur et où le pipeline graphique pourrait écrire les résultats. Ainsi, nous définissons un framebuffer FB comme :

```
type FB = (Ar I (FBO C), [FBO Dpth])
```

Le premier élément de ce tuple est un array des FBO dont le type est la couleur C. Puisque un array doit contenir au moins un élément, son utilisation ici est convenable. En revanche, le deuxième élément est défini par une liste qui peut être soit vide, soit contenir exactement un élément FBO dont le type est la profondeur Dpth. Cette liste exprime la possibilité, mais pas l'obligation, de connecter au framebuffer la texture de profondeur.

Un pixel est le résultat du traitement des fragments et il sera inscrit dans les objets du framebuffer exprimant la couleur : (FBO C). Le type d'un pixel va ainsi contenir au moins une information sur la couleur. Il peut aussi contenir plusieurs informations sur la couleur si le framebuffer contient également plus d'un objet (FBO C). C'est, en effet, le cas du rendu dans plusieurs textures. Un pixel contient également une information sur la position 2D, exprimée par le type P. En contraste des fragments, il ne contient aucune information sur la profondeur car il s'agit d'un point coloré dans l'image. Un pixel PX est définit comme :

```
type PX = (P, Ar I C)
```

4.5.1.8 Type pour l'environnement de travail

Le pipeline graphique correspond à une architecture du calcul. Il est entouré, parmi d'autres, par la mémoire qui nous sert à stocker les images. L'environnement de travail du pipeline graphique est défini pour les besoins de notre traitement par le type Env et nous y incluons le framebuffer FB et les textures TX exprimés par une liste. Cette liste peut contenir plusieurs textures mais peut être vide dans le cas où nous ne travaillerions pas avec les textures.

```
type Env = (FB, [TX])
```

Pour pouvoir ré-inclure les résultats de traitement perçus comme un nouveau framebuffer FB, nous définissons une fonction refreshFB de mise à jour du framebuffer dans l'environnement Env :

```
refreshFB :: Env → FB → Env
refreshFB (_, txs) fb = (fb, txs)
```

qui insère le framebuffer, passé par l'argument *fb*, dans l'environnement sortant de cette fonction.

Nous utiliserons par la suite la fonction de création de l'environnement mkEnv, cf. définition exacte en Annexe B, page 202. Et nous utiliserons aussi deux fonctions de manipulation, getTXs et getFB, dont les définitions exactes sont présentées également en Annexe B, page 202.

4.5.1.9 Les commandes graphiques

Les commandes passés au GPU, exprimées par le type Commands, ont la forme des primitives graphiques, exprimés par le type Shape qui spécifie le modèle de la forme. Ce modèle est complété par les

données descriptives, les vertex V. Chaque forme a un nombre défini des vertex associés qui donnent à la forme des valeurs concrètes et la définisse précisément. On inclut dans les vertex également les informations supplémentaires telles que la couleur, les coordonnées de la texture liée avec cette forme, dans le graphisme 3D il pourrait s'agir des paramètres de la surface, etc.

```
type Commands= ([Shape],[V])
```

4.5.2 Primitive de calcul avec le pipeline graphique

Pour formaliser le fonctionnement du pipeline graphique, nous allons définir quelques fonctions qui utiliseront les types définis préalablement et vont donner les correspondants formels aux blocs opérationnels du pipeline graphiques et à la façon de leur fonctionnement. Le tableau 4.2 présente une liste complète de ces fonctions.

Nom de la fonction	Nom du type	Désignation	Signature de type
—	Sampler	Échantillonage des textures	(TX → TXP → C)
vprocessor	—	Processeur des vertex	(Env → VProg → [V] → [V])
—	VProg	Vertex programme	(Env → V → V)
fprocessor	—	Processeur des fragments	(Env → FProg → [F] → [F])
—	FProg	Fragment programme	(Env → F → F)
rprocessor	—	Opérations du framebuffer	(Env → RProg → FB → [F] → FB)
—	RProg	Raster programme	(Env → FB → F → FB)

TAB. 4.2 : Signatures de type des primitives du calcul du pipeline graphique et les GPU

4.5.2.1 Échantillonnage des textures

L'échantillonnage des textures est une des opérations utilisées dans deux unités - l'unité de traitement des vertex et l'unité de traitement des fragments. Elle se présente à l'utilisateur par les *samplers*, les blocs configurables d'accès à la mémoires des textures. Dans cette thèse, nous définissons les *samplers* comme des fonctions du type Sampler qui, pour une texture TX et une position TXP données, extraient une information à partir de cette texture. Le résultat d'échantillonnage se présente par le vecteur de couleurs de sortie C.

```
type Sampler = (TX → TXP → C)
```

Le fonctionnement exact des fonctions d'échantillonnage est dépendant des capacités matérielles des processeurs graphiques et est configurable selon nos besoins. Ces capacités sont largement suffisantes pour notre travail et nous n'en allons utiliser que certaines configurations.

La première des fonctions d'échantillonnage que nous allons utiliser en la morphologie mathématique est la fonction smpBorder qui nous retourne les points de la texture tx correspondant à la coordonnée tp si cette coordonnée est présente dans l'étendue d'indexation de la texture, sinon, elle nous retourne la valeur de bord associée à la texture.

```
smpBorder :: Sampler
smpBorder tx tp | inbounds2D(bounds$ar) tp = ar!tp
                | otherwise               = getTXBFromTX$tx
    where ar = getArFromTX$tx
```

4.5.2.2 Traitement des vertex

Les vertex sont traités dans une unité de traitement que nous appelons le *processeur des vertex* et qui est définie dans notre formalisme fonctionnel par la fonction vprocessor. Son fonctionnement exact est

décrit par un programme VProg qui est exécuté sur chaque vertex du stream d'entrée. Ce programme peut obtenir certains paramètres globaux à partir de l'environnement de travail Env ou il peut obtenir des données à partir des textures (stockées dans l'environement Env).

```
vprocessor           ::  Env → VProg → [V] → [V]
vprocessor e vp vs = map (vp $ e) vs
```

Le *vertex programme* lui-même est défini comme une fonction du type VProg qui est connectée à l'environnement Env et qui effectue une opération sur un vertex V. Le type V du vertex constitue également le type de sa valeur de sortie. Nous ne donnons ici que la signature de type VProg du *vertex programme* car c'est ce programme qui exprime la capacité de programmation des GPU et nous allons le définir spécifiquement pour chacun de nos algorithmes.

```
type VProg = (Env → V → V)
```

Comme un exemple trivial d'un *vertex programme*, nous présentons la fonction vpid d'identité qui ne modifie pas le vertex d'entrée et le retourne aussitôt :

```
vpid    ::  Env → V → V
vpid e v = v
```

4.5.2.3 Rastérisation des primitives géométriques

Un autre bloc fonctionnel dans le pipeline graphique est le rastériseur. Il s'agit de l'unité qui crée les fragments à partir des données décrivant une forme géométrique. Nous le définissons formellement comme fonction du type Rasterizer qui est connectée à l'environnement Env et qui prend comme argument un tuple composé du flux des valeurs de la forme géométrique et du flux de vertex, ([Shape],[V]). Les fonctions de ce type extraient, selon les valeurs concrètes décrivant la forme, un nombre défini de vertex [V]. La sortie de ce bloc est, bien sûr, le flux des fragments [F] dérivés de cette forme.

```
type Rasterizer = (Env → ([Shape],[V]) → [F])
```

Nous ne donnons à cette place que la signature de type Rasterizer d'un rastériseur. C'est dû au fait que le rastériseur est également une unité configurable. Par conséquent, les fonctions de ce type seront définies dans nos algorithmes et elles vont correspondre à la configuration particulière de cette unité.

4.5.2.4 Traitement des fragments

L'unité de traitement des fragments, appelée *processeur des fragments*, est définie d'une façon similaire au *processeur des vertex* (dans 4.5.2.2, page 81). Le processeur des fragments est modelé par la fonction fprocessor qui exécute un fragment programme FProg sur un flux d'entrée des fragments [F]. Ce processeur peut obtenir des données supplémentaires à partir de l'environnement de travail Env ce qui se présente comme la capacité d'échantillonner les textures. Le processeur de traitement des fragments a pour résultat également un flux des fragments [F].

```
fprocessor           ::  Env → FProg → [F] → [F]
fprocessor e fp fs = map (fp $ e) fs
```

Le *fragment programme* est une fonction du type FProg :

```
type FProg = (Env → F → F)
```

Nous ne présentons que la signature de type pour les *fragment programmes*. Les fonctions seront définies spécifiquement pour chaque algorithmes en assurant la fonctionnalité convenable à notre cas d'utilisation. Le programme trivial d'un *fragment programme* est le programme d'identité fpid qui ne modifie rien sur le fragment d'entrée et le retourne inchangé aussitôt.

```
fpid    ::  Env → F → F
fpid e f = f
```

4.5.2.5 Opération du framebuffer

Les opérations du pipeline graphique qui sont regroupées dans notre diagramme de blocs, fig. 3.18, page 52, sous le nom *Raster opérations* constitue, en effet, un bloc fonctionnel tout entier qui traite les fragments, les convertit en pixels et se charge de leur fusion avec le contenu déjà présent dans le framebuffer.

La manière dont l'information d'un fragment est fusionnée avec les données du framebuffer peut être configurée par l'utilisateur et nous pouvons, si notre matériel dispose de telles capacités, avoir également les fonctionnalités de post-traitement des pixels plus ou moins complexes.

Nous avons regroupé toutes ces opérations dans un bloc que nous avons nommé *raster processor* mais qui ne correspond pas sur les GPU à un vrai processeur mais plutôt à un certain nombre des blocs fonctionnels qui sont enchaînés dans un pipeline et dont la fonction peut être activée par l'utilisateur.

Pour suivre la même logique que pour le *vertex processeur* et le *fragment processeur*, nous définissons le raster processeur par la fonction rprocessor. Elle prend un programme RProg comme paramètre et englobe tous les blocs de post-traitement travaillant avec les informations du framebuffer dans une seule fonction du Haskell :

```
rprocessor          :: Env → RProg→ FB → [F] → FB
rprocessor e rp fb fs = foldl (rp$e) fb fs
```

Ce processeur peut obtenir des paramètres de configuration de l'environnement Env et applique le programme RProg sur tous les fragments du stream d'entrée [F] en utilisant les données du framebuffer FB. Le *raster programme* se charge également de l'écriture d'un nouveau pixel issu de ces opérations dans le framebuffer. La fonction du Haskell **foldl** qui est utilisée ici est parfaitement convenable pour notre travail. Elle correspond à la réduction du stream par une fonction dont les arguments sont de deux types différents et qui est parfaitement convenable pour notre travail.

Nous présentons ici la signature de type RProg du *raster programme*, la définition précise sera spécifiée ultérieurement selon les besoins particuliers de nos algorithmes.

```
type RProg  :: (Env → FB → F → FB)
```

4.5.3 Modèle du pipeline graphique des GPU

Nous assemblons un modèle mathématique du pipeline graphique à partir des primitives du calcul que l'on vient de présenter. Ainsi, le pipeline graphique est défini comme fonction pipeGPU qui enchaîne dans une séquence de traitement le *vertex processeur* vprocessor, *rastériseur* ras, *fragment processeur* fprocessor, et les opérations du framebuffer exprimées par un processeur abstrait rprocessor. Un nouveau framebuffer issu de notre calcul est incorporé dans l'environnement de sortie de ce pipeline par la fonction refreshFB.

```
pipeGPU :: VProg → Rasterizer→ FProg → RProg→ Commands → Env → Env
pipeGPU vp ras fp rp (ss, vs) e =
    (refreshFB e)
  ∘ (rprocessor e rp (getFB e))
  ∘ (fprocessor e fp)
  ∘ (ras e)
  $ (ss, (vprocessor e vp) vs)
```

Le comportement de ce pipeline est modifiable par les fonctions passées comme arguments de ce pipeline. VProg définit le programme du *vertex processeur*, ras définit la manière exacte de rastérisation des formes géométriques, FProg définit le programme du *fragment processeur*, RProg définit le fonctionnement des opérations sur le framebuffer et la manière dont les fragments sont transformés en pixels et dont les pixels sont fusionnés avec les données dans le framebuffer.

4.6 Primitives de la morphologique mathématique

Les méthodes de la morphologie mathématiques auxquelles sont destinés nos skeletons algorithmiques et les algorithmes dérivés de ces derniers sont déjà bien décrites dans plusieurs publications scientifiques. Nous pensons que les définitions de base, des propriétés des opérateurs morphologiques et les explications standard de leur fonctionnement sont bien connus et il n'est pas donc nécessaire d'inclure une introduction détaillée à la morphologie mathématique. Nous adressons le lecteur aux publications qui sont consacrées à l'explication des bases de la morphologie mathématique. Nous recommandons un article de Henk Heijmans[Hei95] qui présente sur quelques pages une introduction aux principes de base de la morphologie mathématique. Pour une introduction plus profonde, nous recommandons un livre récent de Pierre Soille[Soi03] qui familiarisera le lecteur avec les opérations morphologiques le plus utilisées dans la pratique. Pour une explication systématique et les concepts avancés de la morphologie, nous recommandons les livres de Jean Serra[Ser88, Ser89] comme référence.

Dans la suite, nous ne présenterons que les définitions qui bâtissent les briques de bases et qui seront réutilisées dans nos prochaines définitions des algorithmes. Nous allons présenter les opérations morphologiques dans le formalisme fonctionnel qui n'est pas habituellement utilisé pour telles définitions mais qui fait un très bon lien entre les définitions mathématiques classiques, comme présentées dans les sources bibliographiques introductives que nous venons de mentionner, et les implémentations informatiques de ces définitions par la programmation, fonctionnelle dans notre cas.

4.6.1 Images dans la morphologie mathématique

Dans la morphologie discrète, nous travaillons avec des images discrètes où c'est le domaine de ces dernières mais également les valeurs de leurs pixels qui sont digitalisés. Nous mentionnons cela tout en sachant qu'une image peut avoir plus qu'une seule valeur numérique jointe à un pixel, ce qui est le cas, par exemple, pour les images multispectrales où les images dont les valeurs sont les vecteurs paquetés.

Ainsi, nous définissons une image I comme une fonction définie sur un certain domaine $D \subset \mathbb{Z}^n$, $n \in \mathbb{N}$ et dont les valeurs sont définies par un ensemble $V \subset \mathbb{Z}$:

$$I\{ \begin{array}{l} D \to V \\ p \mapsto I(p) \end{array} \tag{4.1}$$

Cette définition est générale et ne définit pas les détails sur la forme exacte du domaine ni sur l'ensemble des valeurs. Pourtant, dans les applications pratiques travaillant avec les images des caméras vidéo, nous choisissons souvent le domaine d'une image 2D comme un array 2D. Cette interprétation mène à la forme que nous utilisons dans cette thèse pour l'expression d'une image en tant qu'array en formalisme fonctionnel, cf. 4.3.1.4, page 65 :

```
Ar (I,I) α
```

où le domaine D de la définition correspond à la classe des index **Ix** dans Haskell que nous spécialisons, pour nos besoins, à un tuple pour une image 2D.

4.6.2 Grilles et voisinages

Pour exprimer les relations de connexité entre les éléments d'une image, nous définissons la notion de la grille. Une grille G est définie comme :

$$G \subset \mathbb{Z}^n \times \mathbb{Z}^n, n \in \mathbb{N} \tag{4.2}$$

Nous définissons également la notion du voisinage. Habituellement, le voisinage est défini comme un ensemble des voisins du point p et on définit également le voisinage élargi comme un ensemble des voisins incluant ce point p. Pour des raisons pratiques, il est convenable de distinguer le terme *ensemble*

des voisins du terme *voisinage* et définir le dernier en incluant le point p car c'est avec une telle définition du voisinage que nous allons travailler le plus en pratique.

Ainsi, nous comprenons sous le terme l'*ensemble des voisins* un ensemble des points de l'espace qui sont voisins à un autre point en se basant sur les relations locales définies par une grille donnée. Ceci dit, l'ensemble des voisins $N'_G(p)$ du point p sur la grille G est défini comme :

$$N'_G(p) = \{p' \in \mathbb{Z}^n, (p, p') \in G, n \in \mathbb{N}\} \tag{4.3}$$

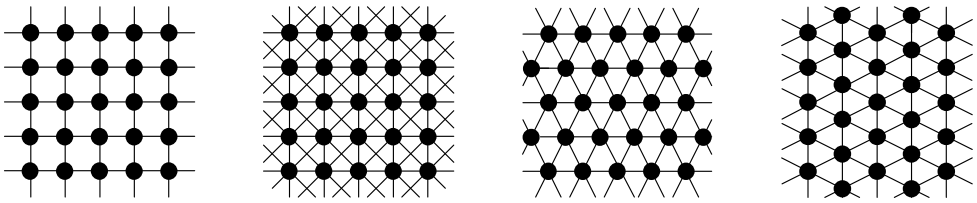
Par extension, $N'_G(A)$ est l'ensemble des voisins d'un ensemble A.

En revanche, le voisinage dit élargi que nous allons appeler désormais *voisinage* est un ensemble constitué du point p et de son ensemble des voisins. Ainsi, le voisinage $N_G(p)$ du point p sur la grille G est définit comme :

$$N_G(p) = \{p\} \cup N'_G(p) \tag{4.4}$$

La grille et la notion du voisinage qui sont associées à l'image (cf. l'équation 4.1) nous définissent les relations entre les pixels et définissent, en utilisant ces derniers, les ensembles des pixels voisins. Pourtant, les équations 4.2 et 4.3 qui sont devenues classiques dans la morphologie ne sont pas restreintes au domaine D de l'image I. Ce qui nous posera des problèmes lors de travail avec les bords de l'image où la grille définit également les relation entre les pixels à l'intérieur du domaine D de l'image I et les index qui ne sont pas inclus dans ce domaine et se trouvent à l'extérieur de l'image. Par conséquent, la définition de l'ensemble des voisins N'_G inclut également les points de l'extérieur du domaine de l'image.

Les grilles le plus souvent utilisées dans les applications pratiques de la morphologie mathématique sont représentées sur le fig. 4.10. Il s'agit notamment de la grille carrée avec 4-connexité entre les pixels (q.v. 4.10(a)) et de la grille carrée 8-connexe (q.v. 4.10(b)). D'autres types importants de grilles, largement utilisés dans la morphologie mathématique pour leurs propriétés de connexion lors du travail avec les composantes connexes (cf. l'article [Soi03] pour plus de détails à ce sujet), sont appelées les grilles *hexagonales*[1]. La grille la plus souvent utilisée dans les applications qui traitent des signaux vidéos est celle illustrée par la fig. 4.10(c) que nous appelons *décalée par lignes*. Mais il est possible d'envisager l'utilisation d'une grille décalée par les colonnes, cf. 4.10(d), qui n'est pas couramment employée mais à laquelle nous devons faire appel si nous travaillons avec les images que nous transposons à l'intérieur de nos algorithmes.



(a) Grille carrée 4-connexe

(b) Grille carrée 8-connexe

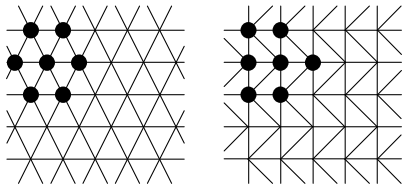
(c) Grille hexagonale 6-connexe, décalée par lignes

(d) Grille hexagonale 6-connexe, décalée par colonnes

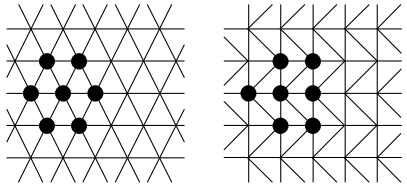
FIG. 4.10 : Grilles utilisées dans la morphologie mathématique

Le problème majeur avec les grilles hexagonales est qu'elles emploient les positions dans l'images qui ne sont pas des nombres entier, car la ligne/colonne est décalée de 0.5 de la distance entre les pixels. Ce qui n'entre pas en cohérence avec les définissions décrites précédemment pour la grille (cf. équation 4.2) où nous exigeons les nombres entiers. Ce problème est encore plus marquant lors du stockage des pixels dans la mémoire à l'aide d'un array 2D qui est décrit, en effet, par la grille carrée.

[1] Il s'agit, en effet, de grilles triangulaires, la notion d'hexagone surgit lors du travail avec le voisinage



(a) sur les lignes avec index $2i$



(b) sur les lignes avec index $2i+1$

FIG. 4.11 : Voisinage définit sur une grille hexagonale avec 6-connexité (décalée par lignes) et sa transposition à une grille carré avec 6-connexité (décalée par lignes)

C'est pourquoi nous définissons les grilles carrées avec 6-connexité qui font une correspondance entre le stockage de données dans une grille carrée (dans un array) mais dont le l'ensemble des voisins de chaque point désigne les points de la grille hexagonale correspondante. La figure 4.11 nous montre cette mis en correspondance entre la grille carrée et la grille hexagonale et le voisinage non symétrique qui est défini différemment sur les lignes paires et impaires. La figure 4.12 démontre le même principe pour le travail avec les grilles décalées par colonnes.

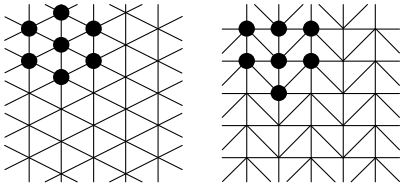
4.6.3 Éléments structurants

Les éléments structurants sont définis comme un sous-ensemble d'un espace vectoriel, pas nécessairement celui définissant la grille. Donc, un élément structurant est défini par un ensemble des vecteurs de déplacements qui, utilisés pour les opérations de Minkowski, sont employés à déplacer l'ensemble (image) pour y ensuite appliquer une opération ensembliste, cf. fig. 4.13(a) qui présente l'addition de Minkowski.

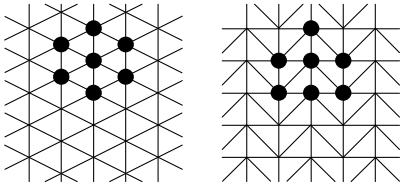
Les opérations morphologiques de base, la dilatation et l'érosion, sont définies[1] par les opérations de Minkowski, mais en utilisant un élément structurant transposé. La différence entre les deux opérations est percevable après avoir comparé la fig. 4.13(b) (pour la dilatation et l'élément structurant transposé) avec la fig. 4.13(a) (pour l'addition de Minkowski et l'élément structurant dans sa version non-transposée).

Notons que si nous utilisons les approches qui ne travaillent pas à l'échelle des images mais travaillent à l'échelle des pixels à l'aide des kernels et en utilisant l'extraction des éléments à partir des vecteurs de déplacements, nous ne pouvons pas, pour le calcul des opérations morphologiques de base, utiliser directement les vecteurs définis par l'élément structurant transposé. En effet, lors des extractions des voisins pour le calcul de la dilatation par un kernel à l'échelle des pixels, nous utilisons les vecteurs de déplacement qui ne correspondent pas à ceux qui ont été donnés pour la dilatation mais à leur version transposée. Ainsi, nous avons recours à double transposition (une pour la dilatation, une pour l'extraction à l'échelle d'un kernel). C'est-à-dire, les vecteurs de déplacement que nous utilisons lors de l'extraction des voisins sont ceux définis par l'élément structurant non-transposé. La fig. 4.13(c) illustre ce raisonnement.

[1] Plus de détail sur les définitions de la dilatation et de l'érosion par les opérations de Minkowski, cf. page 43 du livre [Ser89] de Serra

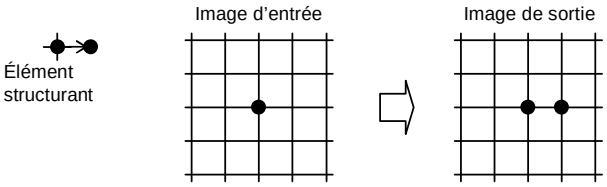


(a) sur les les colonnes avec index $2i$

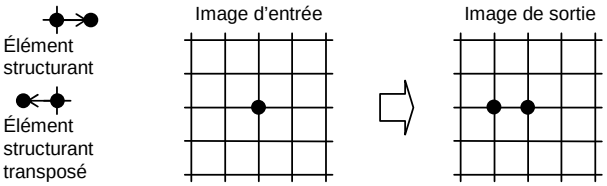


(b) sur les colonnes avec index $2i + 1$

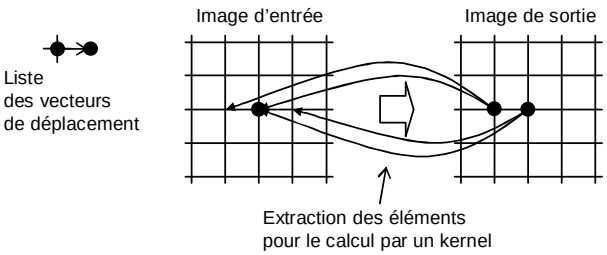
FIG. 4.12 : Voisinage défini sur une grille hexagonale avec 6-connexité (décalée par colonnes), deuxième type et sa transposition à une grille carré avec 6-connexité (décalée par colonnes)



(a) Élément structurant et l'addition de Minkowski



(b) Élément structurant transposé et la dilatation



(c) Liste des vecteurs de déplacement pour l'extractions des éléments lors du travail avec les kernels

FIG. 4.13 : L'utilisation des éléments structurants dans les opérations morphologiques et les listes des vecteurs de déplacement lors du travail avec les kernels

Dans notre approche fonctionnelle, les notions du voisinage et de l'élément structurant (transposé ou non-transposé) sont très proches car les deux sont exprimées par la même structure de données : la liste du Haskell. Dans les implémentations des opérations morphologiques qui vont utiliser la fonction d'extraction des pixels, nous allons travailler avec les listes des vecteurs de déplacement. Ces listes perdent l'information syntaxique et en les utilisant, nous n'allons pas faire une distinction explicite entre un élément structurant en sens large (transposé ou non-transposé) et entre le voisinage d'un pixel. Le sens exact de ce que cette liste représente sera donné par la définition de l'opération à effectuer.

Ainsi, nous définissons le type Ngb pour pouvoir exprimer la liste de ce type.

```
type Ngb        = [(I, I)]
```

Comme exemple des définitions de ces listes, nous montrons la définition du *voisinage* de la grille carrée 4-connexe, ngbSQR4 :

```
ngbSQR4  = [(0,0), (0,1), (1,0), (0,−1), (−1,0)]  :: Ngb
```

et la manière dont nous dérivons l'*ensemble des voisins* ngbSQR4WC à partir de ce voisinage :

```
ngbSQR4WC  = ngbSQR4\\[(0,0)]  :: Ngb
```

Les listes de déplacement utilisées sur la grille hexagonale (décalée par lignes) sont définies en se basant sur une ligne donnée (ici la ligne avec un index pair) comme, par exemple :

```
ngbSQR6  = [(0,0), (0,1), (1,0), (1,−1), (0,−1), (−1,−1), (−1,0)]  :: Ngb
```

Ce qui correspond au voisinage illustré sur la fig. 4.11(a). La forme pour le voisinage sur la ligne impaire est soit définie exactement pour la ligne impaire, soit déduite de l'élément structurant pour la ligne paire par décalage des index dans la fonction d'extraction du voisinage, comme nous le verrons dans la section suivante.

4.6.4 Extraction du voisinage

4.6.4.1 Concrétisation des index des pixels désignés par l'élément structurant

Dans la suite, nous allons utiliser les fonctions pour l'obtention des index concrets constituant le voisinage local d'un élément. Les index ne seront pas restreints sur le domaine de l'image, c'est-à-dire, il peuvent désigner un index au-delà du domaine de l'image.

Ces fonction seront du type SpecNgb :

```
type SpecNgb  = Ngb → (I,I) → [(I,I)]
```

qui vont nous livrer, en se basant sur la liste des déplacement relatifs et pour une position dans l'image concrète, (I, I) la liste des index de déplacement concrets.

Nous définissons la fonction specNgbSQR qui, étant spécifique pour la grille carrée, prend la liste des index de déplacement *ngbs* et qui, pour un index d'un élément donnée (x, y), nous retourne les index spécifiques des voisins de ce pixel ou des éléments désignés par l'élément structurant de ce pixel :

```
specNgbSQR              :: SpecNgb
specNgbSQR ngbs (x,y)   = map ( λ(dx,dy) → (x+dx, y+dy)) ngbs
```

De la même manière, nous pouvons définir d'autres fonctions pour d'autres grilles. Pour illustrer la construction d'une telle fonction pour la grille hexagonale décalée par lignes (parallèlement à la deuxième coordonnée), nous présentons la définition de la fonction specNgbHEXR qui est plus complexe :

```
specNgbHEXR              :: SpecNgb
specNgbHEXR  ngbs (x,y) | even x    = map ( λ(dx,dy) → (x+dx, y+dy)) ngbs
                         | otherwise=fncngbs(x,y) [ ]
    where
        fnc ((dx,dy):ngbs) (x,y) res | even dx =  fnc ngbs (x,y) ((x+dx, y+dy):res)
        fnc ((dx,dy):ngbs) (x,y) res | odd  dx =  fnc ngbs (x,y) ((x+dx, y+dy+1):res)
        fnc [ ] _ res = res
```

Cette fonction est spécialisée pour le travail avec le voisinage défini, par convention, sur la ligne paire de la grille carrée équivalente à la grille hexagonale décalée par lignes (eg. ngbSQR6).

4.6.4.2 Traitement de l'extérieur du domaine fini de l'image

Le traitement des valeurs des voisins ou des valeurs définies par un élément structurant et qui n'appartiennent pas au domaine de l'image mais à son extérieur constitue l'activité la plus problématique du traitement par les kernels du calcul. Ce qui nous pose des problèmes ici c'est le fait que nous n'avons pas un élément de l'image correspondant aux index de déplacement définissant les voisins en-dehors du domaine fini de l'image. Par conséquent, nous sommes obligés d'utiliser des approches particulières pour l'extraction des valeurs à partir des index qui se trouvent à l'extérieur de l'image. Un certain nombre de techniques existe et leur utilisation précise est déterminée par l'application pratique. L'approche utilisée le plus souvent est celle de la valeur du bord constante.

Pour pouvoir traiter les valeurs extérieurs du domaine de l'image, nous définissons un type BorderFnc de fonctions qui vont se charger de nous retourner, pour un array donnée et pour une position concrète (I, I) n'appartenant pas au domaine de l'image, une valeur précise :

```
type BroderFnc α = (Ar (I,I) α → (I,I) → α)
```

La fonction cBorder qui travaille sur les valeurs scalaires α nous retourne toujours la constante val indépendamment de la valeur concrète de l'index pos.

```
cBorder :: (Ord α) ⇒ α → BroderFnc α
cBorder val = (λ ar pos → val)
```

Pour le traitement SIMD, la définition de la fonction cBorderSIMD percevant les bords comme les éléments paquetés de la valeur constante est un peu différente mais elle assure la fonctionnalité du même type

```
cBorderSIMD :: PVec I α → BroderFnc(PVec I α)
cBorderSIMD val = (λ ar pos → val)
```

D'autres fonctions de gestion particulière de l'extérieur de l'image peuvent être envisagées est définies de la même manière. Notons que vu le caractère local des opérations dans la morphologie, nous parlons à l'occasion du traitement des index extérieurs au domaine de l'image également du *traitement des bords* ou de la *gestion des bords* de l'image.

4.6.4.3 Échantillonnage

Pour assurer l'extraction du voisinage à partir des éléments de l'image, les accès à la mémoire sont nécessaires. Pourtant, si les index désignés par l'élément structurant ne sont pas du domaine de l'image, nous devons faire appel à une quelconque technique de gestion des bords, comme on vint de le décrire dans la section précédente.

Il semble convenable de formaliser à l'échelle des fonctions du Haskell l'obtention des valeurs correspondant aux index concrets. Ce processus va intégrer dans un seul type de fonctions les fonctionnalités d'indexation des éléments à l'intérieur de l'array et les fonctionnalités de traitement des valeur à l'extérieur du domaine de ce dernier. Nous allons appeler ces fonctions sous un terme commun *échantillonnage*. Ces fonction seront du type SampFnc :

```
SampFnc α :: (Ix β) ⇒ Ar β α → β → α
```

La fonction d'échantillonnage est une fonction qui nous retourne, pour un index concret du type β, une valeur qui est du type α – du même type que les éléments de l'image. Soit elle va incorporer le traitement des valeurs de bords déjà dans sa définition, soit, dans le cas où nous savons explicitement que nous n'allons pas indexer les éléments au-delà du domaine de l'image, elle n'a pas besoin d'assurer le traitement des bords et par conséquent, elle n'est pas obligée d'incorporer une telle fonctionnalité.

Démontrons quelques définitions de ces fonctions qui seront utilisées par la suite dans les descriptions des algorithms. La fonction d'échantillonnage sampl qui n'est spécialisée qu'à l'indexage des éléments à l'intérieur du domaine de l'image est définie comme :

```
sampl α          :: (Ix β) ⇒ Ar β α → β → α
sampl ar x = ar!x
```

Remarquons que sa signature de type est compatible avec celle du type type SampFnc :

```
sampl α          :: SampFnc α
```

La fonction sampB ne se spécialise qu'au traitement des bords et elle est définie comme :

```
sampB α          :: (Ix β) ⇒ BroderFnc α → Ar β α → β → α
sampB brdfnc ar x = brdfnc ar x
```

Remarquons que sa signature de type est compatible avec la suivante qui utilise le type SampFnc :

```
sampB α          :: BroderFnc α → SampFnc α
```

C'est-à-dire, elle prend une fonction traitant le bord comme paramètre et nous retourne la fonction d'échantillonnage.

La fonction sampGen assure le cas général où nous pouvons échantillonner avec les index concrets inclus dans le domaine de l'image mais également avec les index qui n'y sont pas inclus. Nous réutilisons les deux fonctions précédentes :

```
sampGen α          :: (Ix β) ⇒ BroderFnc α → Ar β α → β → α
sampGen brdfnc ar x = λx → if inRange(bounds $ ar) x then
                              (sampl ar x)
                           else
                              (sampB brdfnc ar x)
```

Remarquons que sa signature de type est compatible avec celle qui utilise le type SampFnc :

```
sampGen α          :: BroderFnc α → SampFnc α
```

4.6.4.4 Généralisation de l'extraction du voisinage

Nous appelons les *fonctions d'extraction du voisinage* les fonctions qui, pour un array donné et un index donnée, extraient les valeurs des éléments voisins. Elle seront du type ExtrNgb qui est défini comme :

```
type ExtrNgb α = Ar (I,I) α → (I,I) → [α]
```

Cette façon de voir les fonctions pour l'extraction du voisinage est assez générale pour pouvoir inclure tous les cas possibles car elle ne se base pas, dans sa définition, sur une liste concrète des voisins et ne s'occupe pas à l'échelle de ses paramètres par la gestion particulière des voisins dépassant les bords de l'image.

En effet, c'est le corps de la fonction qui définira tous ces détails. Les fonctions que nous définirons par la suite seront plus spécialisées, e.g. pour la grille hexagonale ou la grille carrée, et auront la signature de type différente. Mais après l'application partielle à des paramètres concrets, nous obtiendrons une nouvelle fonction, qui sera du type ExtrNgb des fonctions d'extraction du voisinage.

4.6.4.5 Extraction du voisinage des types de base

L'extraction du voisinage des types de base est assurée par l'utilisation de la fonction de concrétisation des indexes, qui est du type SpecNgb, pour une position donnée *pos* et pour les indexes de déplacement donnés *ngbs* en spécifiant la fonction exacte d'échantillonnage qui est du type SampFnc. Commençons par la présentation des définition de certaines fonction d'extraction du voisinage. Pour un travail le plus générique possible d'extraction du voisinage, nous définissons la fonction extrNgb :

```
extrNgb      :: (Ord α) ⇒ SpecNgb→ SampFnc α → Ngb → ExtrNgb α
extrNgb spec sampl ngbs ar pos = map (sampl$ar) (spec ngbs pos)
```

Dans la pratique, nous utiliserons plutôt les fonctions spécialisées pour un type de grille. La définition de la fonction d'extraction du voisinage sur la grille carrée extrNgbSQR est définie comme :

```
extrNgbSQR      :: (Ord α) ⇒ SampFnc α → Ngb → ExtrNgb α
extrNgbSQR sampl ngbs ar pos = map (sampl$ar) (specNgbSQR ngbs pos)
```

Parmi les fonctions spécialisées pour un travail particulier sur le voisinage, nous montrons la définition de la fonction extrINgbSQR qui ne teste pas si l'index du pixel à extraire est inclus dans le domaine de l'image et qui est utilisée pour l'extraction des voisins à l'intérieur du domaine :

```
extrINgbSQR          :: (Ord α) ⇒ Ngb → ExtrNgb α
extrINgbSQR ngbs ar pos = extrNgbSQR sampI ngbs ar pos
```

La fonction dont le fonctionnement est opposé à cette dernière et qui, pour une liste de déplacement donnée, suppose que tous les index sont au-delà du domaine de l'image. Elle appelle, pour chaque index concret, la fonction du bord *brdfnc* :

```
extrBNgbSQR          :: (Ord α) ⇒ BroderFncα → Ngb → ExtrNgb α
extrBNgbSQR brdfnc ngbs ar pos = extrNgbSQR (sampB$brdfnc) ngbs ar pos
```

Suivant la même logique, nous pouvons définir les fonctions d'extraction du voisinage pour les grilles hexagonales (décalées par lignes) extrNgbHEXR, extrINgbHEXR et extrBNgbHEXR en remplaçant l'appel de la fonction specNgbSQR de concrétisation des index relatifs aux index absolus pour la grille carrée par l'appel de la fonction correspondante. Dans le cas d'une grille hexagonale décalée par les lignes il s'agit de la fonction specNgbHEXR.

4.6.4.6 Extraction du voisinage à partir des vecteurs paquetés

Le travail effectué lors d'extraction des voisins dans le cas où les éléments d'un array sont les vecteurs paquetés n'est pas le même que celui d'extraction d'un élément voisin scalaire. Les vecteurs de déplacement d'un élément structurant définissent, en effet, les déplacement en unités scalaires. Pourtant, à l'échelle des vecteurs paquetés, le groupe d'éléments que nous voulons obtenir comme des voisins relatifs aux éléments d'un vecteur paqueté peut se trouver localisé partiellement dans un vecteur et partiellement dans le vecteur voisin.

C'est pourquoi nous définissons la fonction extract d'extraction d'un nouveau vecteur paqueté à partir de deux vecteurs paquetés voisins. Cette fonction correspond sur les architecture multimédia SIMD directement aux instructions.

```
extract :: (Num α)⇒ I → PVec I α → PVec I α → PVec I α
extract off pv1 pv2 = pvec (lo,hi)
                      (   [(i, pv1!(i+off) )   | i ← [lo .. (hi−off)]]
                       ++ [(j, pv2!(j−hi+off)) | j ← [(hi−off+1) .. hi] ]
                      )
  where (lo,hi) = bounds pv1
```

Le premier paramètre de cette fonction définit le décalage *off*, le deuxième et le troisième paramètre désignent deux vecteurs paquetés voisins. Dans le cas spécial où la valeur d'offset *off* est 0, cette fonction nous retourne le premier paramètre *pv*1, dans l'autre cas spécial où la valeur d'offset *off* est égale à la longueur du vecteur paqueté, cette fonction nous retourne le deuxième paramètre *pv*2. Dans le cas de valeur *off* intermédiaire, cette fonction nous retourne, en se basant sur cette valeur, un vecteur paqueté composé à partir des éléments des deux paramètres *pv*1 et *pv*2 relativement cette valeur.

La fonction extract sera utilisée en interne dans la fonction unaLoadSQR de la lecture non-alignée sur la grille carrée qui extrait des voisins d'un array *ar* pour un index concret (x, y) et un vecteur de déplacement concret (dx, dy) ; n est la taille d'un vecteur paqueté et la fonction *brdfnc* nous définit la manière d'extraction des valeurs au-delà de l'image. Le paramètre *how* définit par sa valeur *"Fst"* ou *"Snd"* l'axe de vectorisation de l'array *ar*. fonction

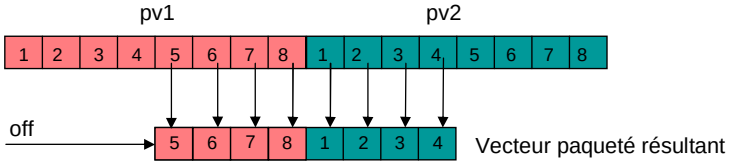


FIG. 4.14 : Illustration du fonctionnement de la fonction extract pour les vecteurs paquetés de 8 éléments et la valeur d'*off* égale à 4

```
unaLoadSQR   :: (Num α)⇒ [Char] → I → SampFnc(PVec I α)
             → Ar (I,I) (PVec I α) → (I,I) → (I,I) → PVec I α
unaLoadSQR how n sampl ar (x,y) (dx,dy)
    | how == "Fst" = extract (mod dx n)
                             (sampl ar (x+(div dx n),y+dy))
                             (sampl ar (1+x+(div dx n),y+dy))
    | how == "Snd" = extract (mod dy n)
                             (sampl ar (x+dx,y+(div dy n)))
                             (sampl ar (x+dx,1+y+(div dy n)))
```

Le processus décrivant la lecture non-alignée sur les grilles hexagonales peut être construit suivant la même logique et en respectant les particularités du travail avec les lignes/colonnes décalées.

Le principe d'extraction des voisins lors du travail avec des vecteurs paquetés est illustré sur la fig. 4.15 pour un exemple concret du voisinage comptant 4 voisins sur la grille carrée dont les vecteurs de déplacement sont définis par ngbSQR4.

Regardons maintenant comment nous mettons tous ces outils pour le travail sur les vecteurs paquetés ensemble. Ayant défini d'une façon générale le type de fonctions d'échantillonnage, la fonction extrNgbSQRSIMD définit l'extraction des voisins des vecteurs paquetés et a le caractère d'un skeleton algorithmique car nous ne donnons pas une prescription exacte pour l'accès aux éléments. Pourtant, l'accès aux éléments est bien spécifié par le type SampFnc et c'est pourquoi nous pouvons l'utiliser dans la fonction de la lecture non-alignée unaLoadSQR sans restreindre à la généralisation. La manière exacte de l'échantillonnage n'est donnée qu'après la spécification de la fonction d'échantillonnage *sampl* par une fonction concrète.

```
extrNgbSQRSIMD   :: (Ord α) ⇒ [Char] → I → SampFnc(PVec I α) → Ngb
                        → ExtrNgb (PVec I α)
extrNgbSQRSIMD how n sampl ngbs ar pos = map (unaLoadSQR how n sampl ar pos) ngbs
```

4.6.5 Kernels de la morphologie mathématique travaillant sur le voisinage local

Une fois les valeurs du voisinage local extraites (ou plus généralement les valeurs des éléments désignés par la liste des vecteurs de déplacement), nous appliquons une fonction locale sur ces valeurs. La fonction est relative à l'opérateur morphologique que nous construisons et sera du type NgbOp :

```
type NgbOp α = ([α] → α)
```

Elle peut être perçue comme un kernel de réduction qui à partir d'une liste des valeurs crée une seule valeur. Regardons alors les exemples des définitions des fonctions.

Le kernel de la dilatation morphologique ngbDilate utilise en interne la fonction foldl1 pour implémenter la réduction par la fonciton max du stream *xs* des valeurs du voisinage local pour évaluer le supremum des valeurs discrètes.

```
ngbDilate    :: (Ord α) ⇒ NgbOp α
ngbDilate xs = foldl1 max xs
```

De la même manière, nous définissons le kernel de l'érosion morphologique ngbErode qui utilise la fonction min comme la fonction par laquelle nous réduisons le stream des valeurs du voisinage local pour évaluer l'infimum des valeurs discrètes.

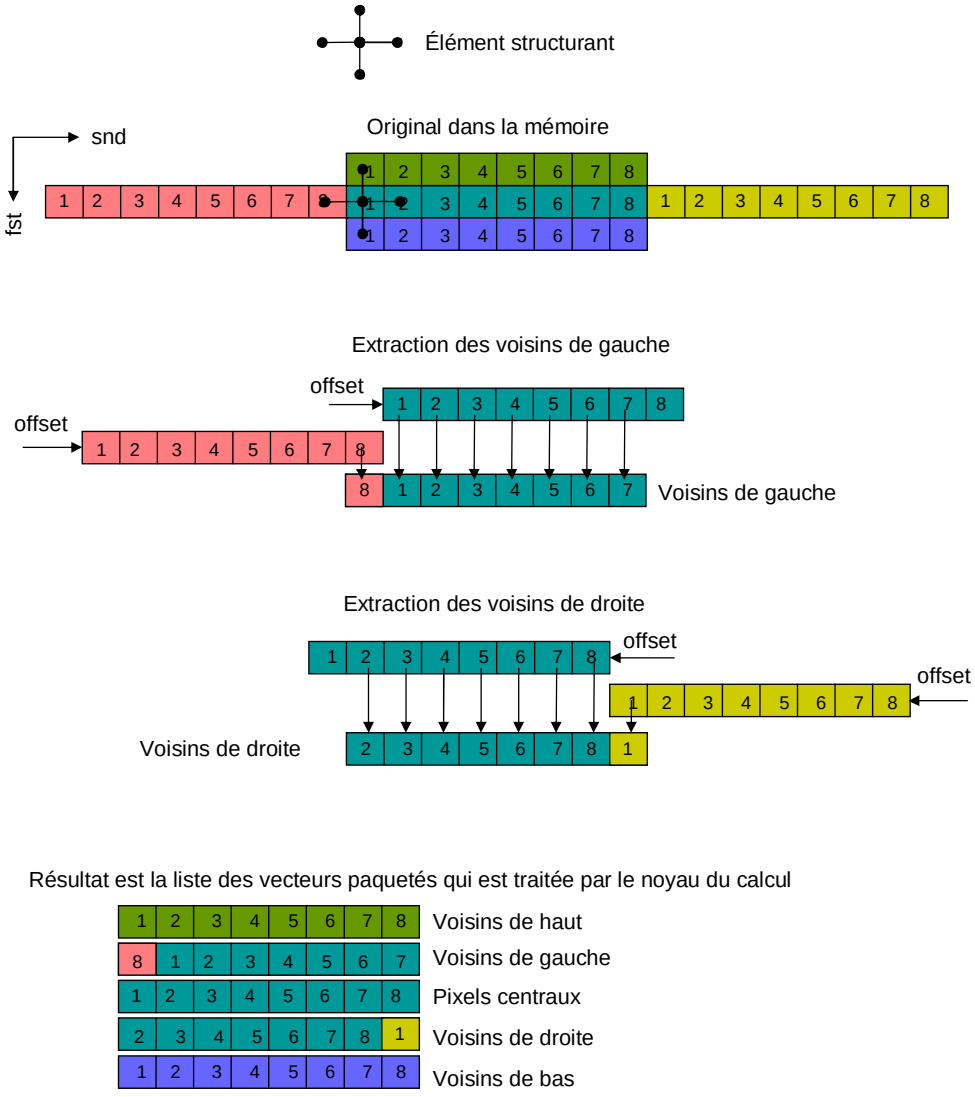


FIG. 4.15 : Extraction des voisins à partir d'un type vector paqueté

```
ngbDilate     :: (Ord α) ⇒ NgbOp α
ngbDilate xs =  foldl1 min xs
```

Pour le travail avec les valeurs SIMD, nous allons utiliser les fonctions maximum et minimum opérant par élément sur les vecteurs paquetés, exprimées respectivement par la fonction maxSIMD :

```
maxSIMD  :: (Ord α) ⇒ PVec I α → PVec I α → PVec I α
maxSIMD  pv1 pv2 = listArray (bounds $ pv1)
                     ( zipWith max (elems$pv1) (elems$pv2) )
```

et par la fonction minSIMD :

```
minSIMD  :: (Ord α) ⇒ PVec I α → PVec I α → PVec I α
minSIMD  pv1 pv2 = listArray (bounds $ pv1)
                     ( zipWith min (elems$pv1) (elems$pv2) )
```

Ces fonctions correspondent directement aux instructions sur les architectures possédant un jeu d'instructions incluant les opérations SIMD. Ainsi, la définition du kernel local de la dilatation morphologique ngbDilateSIMD qui opère sur les vecteurs paquetés est définie comme :

```
ngbDilateSIMD      :: Ord α ⇒ NgbOp (PVec I α)
ngbDilateSIMD xs =  foldl1 maxSIMD xs
```

et le kernel local de l'érosion morphologique ngbErodeSIMD pour les mêmes éléments comme :

```
ngbErodeSIMD      :: Ord α ⇒ NgbOp (PVec I α)
ngbErodeSIMD xs =  foldl1 minSIMD xs
```

4.6.6 Opérations du voisinage local avec un masque

Le travail sur le voisinage local où la propagation des valeurs est restreinte par une fonction du masque ajoute une nouvelle valeur d'entrée pour le traitement. Ainsi, les kernels de calcul seront du type NgbGOp :

```
type NgbGOp α = ([α] → α → α)
```

Par conséquent, les opérations locales géodésiques sont définies comme fonctions qui utilisent en interne les kernels classiques de la morphologie mathématique mais ils appliquent sur leurs résultats la fonctions implémentant la géodésie. Le kernel de la dilatation morphologique géodésique est défini comme ngbGDilate :

```
ngbGDilate      :: (Ord α) ⇒ NgbGOp α
ngbGDilate xs msk =  min msk (ngbDilate xs)
```

et nous définissons, suivant le même principe, aussi la fonction du kernel de l'érosion morphologique géodésique :

```
ngbGErode      :: (Ord α) ⇒ NgbGOp α
ngbGErode xs msk =  max msk (ngbDilatexs)
```

4.6.7 Travail sur le voisinage avec les superpixels

Toutes les techniques de traitement du voisinage local que nous avons présentées (ou de traitement des pixels désignés par la liste des vecteurs de déplacement) sont transposables au travail avec les superpixels, q.v. le concept des superpixels décrit dans 4.4.5, page 73. Pourtant, si nous ne travaillons pas avec les pixels représentés par les éléments de base d'un array mais nous manipulons des entités plus grandes représentées par des groupes des pixels, toutes les fonctions entrant en jeu lors de la définition d'élément structurant, de l'extraction du voisinage et sa représentation en tant que stream et lors du calcul morphologique sur ces streams issus des superpixels seront plus complexes dans leurs fonctionnement.

Ces fonctions ne seront pas, en effet, aussi simples à définir que c'était le cas pour les fonctions travaillant sur les éléments de base. Leurs définitions exactes, qui peuvent être très dépendantes des capacités spécifiques d'une architecture, de la forme de l'élément structurant ou des dimensions de l'image, sont à définir spécifiquement lors de l'implémentation concrète d'un algorithme.

Cependant, les définitions de type de ces fonctions exprimant les entrées et les sorties des kernels de calcul lors de traitement des flux de données sont suffisantes pour pouvoir définir les skeletons algorithmiques. C'est pourquoi nous ne préférons de désigner à cette place les fonctions travaillant sur les superpixels que par leurs signatures de type. C'est cette signature qui sera utilisée pour la description et pour la compréhension de nos skeletons algorithmiques.

Pour pouvoir extraire le voisinage d'un superpixel décrit par son index d'ancrage, nous allons avoir besoin des fonctions qui seront du type ExtrNgbSP :

$$\textbf{type}\ \mathsf{ExtrNgbSP}\ \ \alpha = \mathsf{Ar}\ \ (\mathsf{I},\mathsf{I})\ \alpha \to (\mathsf{I},\mathsf{I}) \to [\alpha]$$

Il s'agit, en effet, de la signature de type qui est compatible avec le type ExtrNgb pour l'extraction du voisinage d'un élément de base. La différence est dans l'information sémantique que porte ce nouveau type ExtrNgbSP et qui nous dit explicitement que nous travaillons avec les superpixels et que l'index donné par (I, I) est l'index d'ancrage d'un superpixel. Elle nous indique également que le stream de sortie peut être large si on le compare avec le stream qui est généré par les fonctions travaillant sur les éléments de base.

Les fonctions qui vont travailler avec ce stream large et vont y appliquer la fonction locale de la morphologie seront désignées par le type NgbOpSP :

$$\textbf{type}\ \mathsf{NgbOpSP}\ \ \alpha = ([\alpha] \to [\alpha])$$

Ces fonctions vont prendre un stream des pixels composant le voisinage d'un superpixel et vont retourner également un stream, celui étant le stream des valeurs constituant le superpixel. Nous mettons cela en contraste avec des fonctions du type NgbOp, cf. 4.6.5, page 92 travaillant sur le voisinage classique qui ne retournent qu'une simple valeur.

Ainsi, nous avons présenté tous les outils nécessaires pour pouvoir décrire les skeletons algorithmiques travaillant sur les superpixels. Ces outils sont représenté par les fonctions de

- pasage d'un array à un stream des index d'ancrage des superpixels (cf. 4.4.5.2) qui sont du type StreamizeSP,
- extraction du voisinage d'un superpixel qui sont du type ExtrNgbSP, comme décrit ci-dessus,
- opération locale sur le voisinage qui sont du type NgbOpSP, comme décrit ci-dessus,
- passage à partir d'une liste des éléments d'un superpixels et de son index d'ancrage aux tuples (index d'élément de base, valeur d'élément de base) qui sont du type ZipSP, cf. 4.4.5.2, page 75.

et nous allons les réutiliser dans la suite de cette thèse dans les algorithmes et les skeletons algorithmiques spécifiques aux superpixels.

Cette page est blanche par intention

Partie II

Algorithmes
et les skeletons algorithmiques

Cette page est blanche par intention

CHAPITRE 5

Algorithmes de voisinage non dépendants du sens du parcours

Ce chapitre sera consacré aux opérations de base de la morphologie mathématique qui travaillent localement sur le voisinage d'un pixel ou sur un autre groupe de pixels défini par l'élément structurant et où les résultats que nous obtenons sont indépendants de l'ordre dans lequel les pixels sont traités. Nous parlons ainsi des opérations indépendantes du sens du parcours de l'image. Il s'agit, en effet, d'un groupe des opérations ensemblistes constituant un des piliers de la morphologie mathématique qui sont largement utilisées dans d'autres algorithmes plus complexes ; cela s'effectue très fréquemment d'une façon répétitive (citons comme exemple [Beu90] les opérations géodésiques, le SKIZ et la ligne de partage des eaux (LPE) par SKIZ). L'implémentation directe de la définition de ces opérations et des opérations dérivées de ces dernières nous conduit aux procédés utilisant abondamment des opérations morphologiques de base et des fonctions logiques.

C'est pourquoi il est primordial de pouvoir exécuter ces opérations le plus efficacement possible sur une architecture donnée. On ressent ce besoin en particulier lors d'un traitement pour les applications en temps réel où le temps du calcul est très précieux et nous ressentons ce besoin d'autant plus si nous utilisons un matériel embarqué pour l'exécution. Dans le cas contraire, nous pourrions nous retrouver avec des temps de traitement non satisfaisants pour un algorithme composé qui fait appel à plusieurs dizaines, centaines ou même à des milliers d'itérations de ces opérations de base.

Nous allons présenter les algorithmes qui décrivent la façon générale du calcul de ces opérations et puis, nous montrerons des approches plus efficaces au calcul de ces opérations sur les architectures avec les capacités SIMD. Plus loin dans ce chapitre, nous allons également présenter les approches qui rendent possible l'évaluation de ces opérations sur les GPU.

Vu que les opérations morphologiques sur le voisinage local entrent entièrement dans la logique du calcul avec les kernels d'exécution, nous transposons leur traitement en traitement en flux de données. Sachant que c'est le principe qui est important, nous n'allons présenter dans la suite que les skeletons algorithmiques de travail sur le voisinage. Un tel skeleton sera exprimé formellement par une fonction du lambda calcul. Les opérations spécifiques de la morphologie mathématique seront obtenues par la spécialisation de fonctionnement de ces skeletons pour des cas d'utilisation particulière.

5.1 Algorithmes élémentaires pour les GPP

5.1.1 Approche naïve à l'implémentation des opérations sur le voisinage

La construction la plus simple d'un algorithme travaillant sur le voisinage nous conduit à l'implémentation que nous appelons *approche naïve*. L'appellation *naïve* est due au fait que nous utilisons la fonction générique pour extraire un pixel et ses voisins, une fonction qui est complexe et qui effectue

pour chaque pixel tous les tests de dépassement de bord de l'image ou de parité de ligne/colonne. Le terme *naïve* n'a pas un sens péjoratif et désigne la méthode la plus simple possible qui peut également être la plus convenable dans la mesure où nous voulions un outil générique et rapidement utilisable et nous ne nous focalisions pas prioritairement sur les performances mais plutôt sur le fonctionnement.

5.1.1.1 *Skeleton algorithmique* ngbAlgo *de l'approche naïve au travail sur le voisinage*

Nous commençons notre explication par la présentation du skeleton algorithmique dédié à la construction des algorithmes concrets travaillant sur le voisinage. Il se décompose en quatre phases. Dans la première, nous passons d'un array 2D à un stream des index ; dans la deuxième, nous extrayons le voisinage local, dans la troisième nous appliquons le kernel travaillant sur le voisinage qui désigne l'opération morphologique et dans la quatrième et dernière phase, nous reconstituons l'array de sortie à partir du stream.

Algorithme 5.1 : ngbAlgo, skeleton algorithmique de l'approche naïve pour le travail sur le voisinage

```
1  ngbAlgo              ::  Streamize α → ExtrNgb α → NgbOp α → Ar (I,I) α → Ar (I,I) α
2  ngbAlgo strm extr op ar = array (bounds ar)
3                            (   (zip ixs)
4                              ∘ (map op)
5                              ∘ (map (extr ar))
6                              $ ixs   )
7          where   ixs = strm$ar
```

L'algorithme 5.1 présente, par la fonction ngbAlgo, la structure de ce skeleton. Il prend quatre arguments dont le premier, *strm* qui est du type Streamize α, désigne la manière dont on parcourt l'image, autrement dit, il s'agit d'une fonction qui retourne un stream des index dans un ordre choisi. Le deuxième, *extr* qui est du type ExtrNgb α, désigne la fonction d'extraction des éléments du voisinage. Cet argument est puissant car très général et nous expliquerons plus tard les éventualités de son utilisation. Le troisième argument, *op* qui est du type NgbOp α, désigne la fonction de l'opération locale sur le voisinage et correspond à une opération morphologique que nous voulons effectuer. Le quatrième paramètre, *ar*, désigne l'array d'entrée qui contient notre image à traiter.

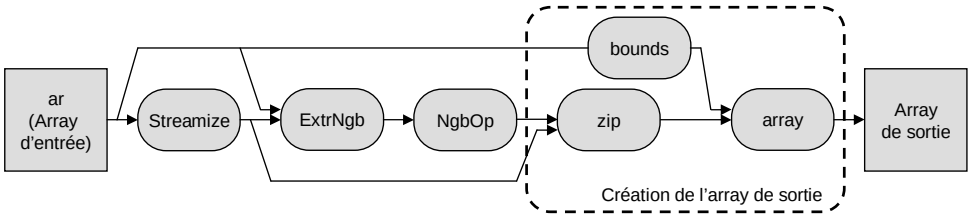


FIG. 5.1 : Graphe de flux exprimant le fonctionnement du skeleton algorithmique ngbAlgo

Le fonctionnement de ce skeleton est assez simple et nous le présentons sur la fig. 5.1 en tant que diagramme de flux. Tout d'abord (ligne 7), nous obtenons un stream des index par l'application de la fonction *strm* sur l'array d'entrée *ar*. Vu que le sens de parcours n'est pas substantiel dans les traitements définis par ce skeleton, la seule obligation posée sur la fonction *strm* est qu'elle doit parcourir toute l'image, l'ordonnancement exact des index dans le stream n'est pas important.

Sur chaque élément de ce stream (ligne 6), nous appliquons sur la ligne 5 (en utilisant la fonction **map**) la fonction *extr* qui représente le kernel d'extraction du voisinage et qui retourne, pour un index donné, une liste des éléments qui constituent le voisinage d'un index donné dans l'array *ar*. Il faut préciser que dans la morphologie mathématique, la structure de cette liste est définie par l'élément structurant,

et elle peut contenir ainsi (ou pas) le pixel même, ses voisins mais également les pixels qui sont plus éloignés des voisins les plus proches du pixel concerné. Malgré ce fait, nous parlons du *voisinage local* (ou de *local neighborhood* en anglais). Et c'est, en effet, la fonction *extr* qui assure tout en ce qui concerne la notion du voisinage, le type de la grille, la manière dont on gère les effets de bord et la manière exacte de l'échantillonnage des pixels dans l'image. Ainsi, nous obtenons un stream large dont les éléments sont les listes composées des pixels formant le voisinage.

Sur tous les éléments de ce stream large, nous appliquons, sur la ligne 4, la fonction **map** qui se charge d'exécuter le kernel de l'opération morphologique *op*. Il s'agit, en effet, d'un kernel de réduction, cf. 3.4.1.2, page 45, qui calcule une nouvelle valeur pour l'index donné à partir d'un ensemble des voisins. Nous obtenons un stream des résultats auquel nous ajoutons (ligne 3) l'information sur la position par la fonction **zip** avec le stream des index *ixs* comme argument. Le résultat de cette opération est un stream des tuples (index, valeur) à partir duquel nous reconstituons, sur la ligne 2, un array de sortie par la fonction standard **array**.

Il faut préciser que nous avons obtenu un skeleton algorithmique qui est très général et dont nous pouvons dériver beaucoup de cas particuliers. Il décrit le principe de fonctionnement et n'exhibe pas explicitement les détails secondaires. Ainsi, la grande partie du travail lourd n'est pas exposée et reste à définir par l'utilisateur lors de la spécialisation de ce skeleton à travers la fonction d'extraction du voisinage ExtrNgb α.

5.1.1.2 *Algorithmes concrets utilisant le skeleton algorithmique* ngbAlgo

Une fois la structure du mode opérationnel de l'approche naïve de travail sur le voisinage présentée, nous allons procéder à la construction des algorithmes concrets et utilisables en pratique par la spécialisation de ce skeleton.

Mais tout d'abord, il faut ajouter une précision concernant l'extraction du voisinage. L'extraction d'un pixel à partir d'un array 2D est simple. Pour pouvoir extraire les valeurs des pixels voisins à ce dernier, nous avons besoin de déterminer certains détails. Pour les applications travaillant sur le voisinage local dans le domaine digitalisé[1], il s'agit notamment :

- du voisinage utilisé – 4, 6, 8 voisins ou autres, représentés par les index relatifs,
- du type de la grille – hexagonale, carrée ou spécifique (graphes de voisinage particuliers), représenté, en général, par la fonction qui met en correspondance les index relatifs désignant les voisins avec l'emplacement exact des voisins dans l'array d'entrée,
- de la façon de gérer les effets de bord – valeur de bord constante, valeur du voisin le plus proche dans l'image, gestion particulière du voisinage aux bords, etc.

C'est la fonction d'extraction du voisinage qui assure toutes ces activités. Elle nous retourne, pour chaque index concret qui localise une position à l'intérieur de l'array, une liste des valeurs des pixels désignées par l'élément structurant. Cette liste, qui est le seul argument d'entrée à l'opération morphologique sur le voisinage, peut être perçue par certains algorithmes comme un stream où l'ordonnancement n'est pas important (e.g. dilatation morphologique), mais nous pouvons utiliser avec avantage la notion de l'ordre dans cette liste dans d'autres algorithmes (e.g. la transformation tout ou rien).

Expliquons cette implémentation (définie par la fonction dilSQR) sur un exemple précis de la dilatation morphologique sur la grille carrée par un élément structurant de 4 voisins. Nous allons utiliser la fonction d'extraction des voisins extrNgbSQR qui est spécialisée pour la grille carrée. Le voisinage est défini par la liste des déplacements relatifs pour un point et ses 4 voisins ngbSQR4. La fonction de traitement des bords que nous avons choisie, cBorder, représente le bord de l'image dont la valeur est constante, c'est-à-dire que cette fonction nous retourne pour tous les index la valeur de son premier argument *brdval*. L'opération de dilatation morphologique est assurée par le kernel de réduction du stream des voisins, ce kernel est exprimé par la fonction ngbDilate. Le dernier point à préciser est l'utilisation

[1] Notons que ce principe est utilisé dans le champs plus large des applications et ne se restreint pas seulement à la Morphologie mathématique

de la fonction standard du Haskell **indices** en tant que fonction de parcours de l'image. En effet, n'importe quelle autre fonction de parcours de l'image pourrait être utilisée dans ce cas car nous créons ici les algorithmes non-dépendants du sens du parcours de l'image.

Voici la définition de la fonction dilSQR :

```
dilSQR :: (Ord α) ⇒ α → Ngb → Ar (I,I) α → Ar (I,I) α
dilSQR brdval ngb ar = ngbAlgo indices (extrNgbSQR (cBorder brdval) ngb) ngbDilate ar
```

De la même manière nous pouvons définir toutes les autres opérations morphologiques dont le fonctionnement n'est pas dépendant du sens du parcours. Par exemple, la dilatation morphologique pour la grille hexagonale et le 6-voisinage, présentée par la fonction dilHEXR :

```
dilHEXR :: (Ord α) ⇒ α → Ngb → Ar (I,I) α → Ar (I,I) α
dilHEXR brdval ngb ar = ngbAlgo indices (extrNgbHEXR (cBorder brdval) ngb) ngbDilate ar
```

Nous pouvons voir que ce n'est que la fonction d'extraction de voisinage (extrNgbHEXR) qui a changé dans cet algorithme par rapport à l'algorithme dilSQR travaillant sur la grille carrée. Le kernel de réduction qui assure l'opération morphologique effectuée sur ce voisinage est resté le même (ngbDilate).

5.1.2 Division du problème en traitement de l'intérieur et en traitement du bord

La première des implémentations plus élaborées de travail sur le voisinage consiste à séparer le traitement de l'intérieur et le traitement du bord. Il s'agit, en effet, d'identifier l'ensemble des index des pixels dont le voisinage est inclus entièrement dans l'image. Pour ces pixels, nous pouvons utiliser l'extraction directe sans nous préoccuper du travail spécial effectuée aux bords de l'image. Les voisins des pixels restants sont ceux qui ont au moins un voisin au-delà des bords et l'algorithme d'extraction des voisins, spécifique pour le traitement des bords, peut y être appliqué.

Quelle est la raison qui nous pousse à effectuer cette division ? Démontrons-la sur l'exemple de la fonction sampGen d'échantillonnage général. Sa définition (q.v. page 90) contient un test conditionnel :

```
λ x → if inRange(bounds $ ar) x then
        (sampI ar x)
      else
        (sampB brdfnc ar x)
```

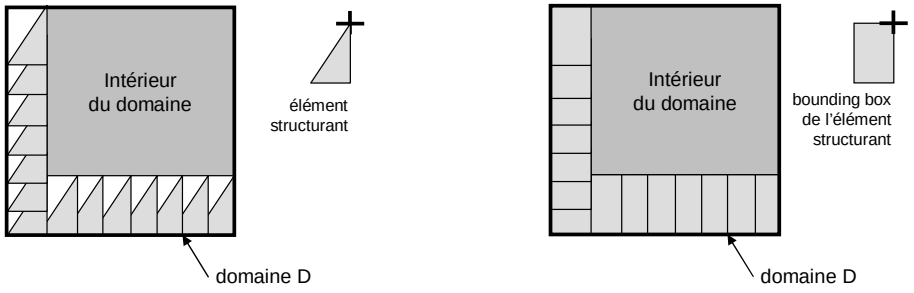
Nous y testons si l'index qui désigne un pixel à extraire est à l'intérieur de l'array. Si tel est le cas, nous procédons à l'échantillonnage de l'image de l'image sampI. Dans le cas contraire, c'est-à-dire dans le cas où l'index pointe au-delà des bords de l'array, nous appelons une fonction d'échantillonnage sampB qui traite les effets de bords par la fonction *brdfnc* avec l'index *x* comme paramètre et qui nous fournit la valeur correspondante. L'idée que nous exploitons en divisant le traitement en deux parties est la suivante : le test qui, dans l'approche naïve, était incorporé dans la fonction d'extraction du voisinage, est déplacé à l'échelle de l'array. Ce que nous faisons c'est la division de l'array à une zone (de l'intérieur) où la condition décrite ci-dessus est toujours valable pour tous les pixels du voisinage et à une zone résiduale (du bord) où cette condition n'est pas remplie pour au moins un index du voisinage.

Il est évident que pour la plupart des éléments structurants utilisés couramment en pratique dont la taille est largement plus petite que les dimensions de l'image, la zone de l'intérieur contiendra une très grande portion des pixels et il est donc préférable de spécialiser le traitement pour cette zone. Le gain que nous pouvons réussir à obtenir avec une telle approche est dépendant du fonctionnement de notre architecture car certaines architectures assurent l'exécution de ce type de condition efficacement et sans délai supplémentaire. Mais il est également évident que s'il faut tester les dépassements du bord pour tous les index d'un élément structurant et pour tous les pixels de l'image avec succès par la condition

```
if inRange(bounds $ ar) x then
```

qui est, pour une donnée *x* de deux dimensions, assurée en général par 4 instructions de test (plus grand / plus petit dans la première / deuxième dimension), il serait plus avantageux de pouvoir éviter ce test, au moins, sur une partie de l'image. Sachant que la majorité des pixels convient à cette condition, toutes

les 4 évaluations élémentaires sont effectuées ici sur un volume important de données et, comme nous le présentons par la suite, ces évaluations peuvent être évitées.



(a) érosion par l'élément structurant

(b) érosion par le bounding box de l'élément structurant

FIG. 5.2 : Identification de l'intérieur du domaine de l'image par l'érosion morphologique, les résultats pour l'élément structurant et son bounding box employé comme élément structurant sont identiques

Trouver l'intérieur du domaine pour n'importe quel élément structurant n'est pas difficile car il correspond au résultat de l'érosion morphologique du domaine par l'élément structurant que nous voulons employer dans l'opération morphologique. De plus, ce résultat est identique à celui de l'érosion du domaine par son *bounding box*. La figure 5.2 illustre cette propriété qui nous permettra de déterminer numériquement et en se basant seulement sur les dimensions de l'image et du bounding box la zone de l'intérieur du domaine dans laquelle l'élément structurant utilisé ne dépasse pas les bords, cf. fig. 5.3.

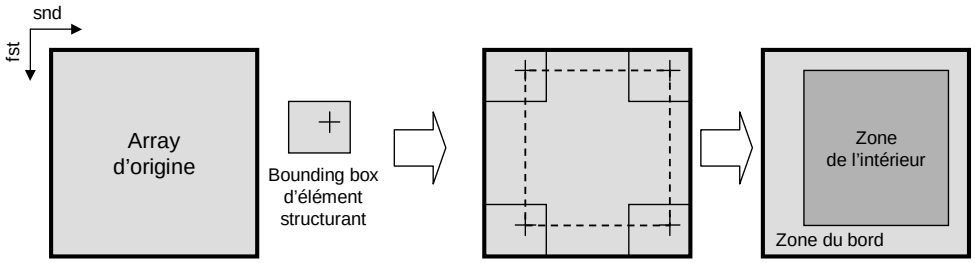


FIG. 5.3 : Division d'un array à la zone de l'intérieur et à la zone du bord

La fonction ngbBB se charge de la détection du bounding box à partir de la liste des déplacements vers les voisins. Le résultat de cette fonction est une tuple des bornes minimales et maximales dans les deux directions.

```
ngbBB :: Ngb → ((I, I), (I, I))
ngbBB ((f,s):xs) = ngbBB xs ((f,s),(f,s))
    where
        ngbBB ((fx,sx):xs) ((flo,slo),(fhi,shi))
                  = ngbBB xs ((min fx flo, min sx slo), (max fx fhi, max sx shi))
        ngbBB [] bb = bb
```

Ainsi, nous pouvons définir une nouvelle implémentation de travail sur le voisinage qui divise le traitement de l'array en deux parties, la partie du traitement de l'intérieur et la partie du traitement du bord. Pour cela, nous divisons l'image en deux zones, en zone de bord et en zone de l'intérieur, comme présenté sur la fig. 5.4.

Le skeleton algorithmique ngbAlgoIB présenté par l'algorithme 5.2 nous montre la structure de ce traitement. Son fonctionnement est déductible à partir de la structure du skeleton algorithmique de l'approche naïve que l'on a présentée dans 5.1.1.1. À la place d'utiliser une seule fonction de parcours de l'image, nous en utilisons deux, une pour la zone intérieure de notre domaine, *strml*, une pour la zone

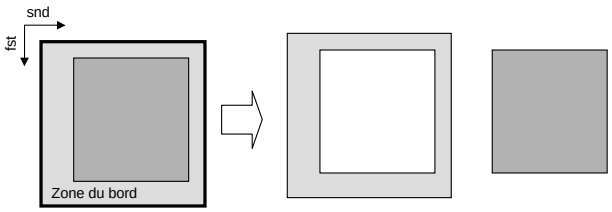


FIG. 5.4 : Décomposition du traitement de l'image en traitement de bord et en traitement de la zone intérieure

du bord de notre domaine, $strmB$. Celles-ci sont représentées dans la signature du type de la fonction ngbAlgoIB par le type Streamize α. Remarquons que pour les algorithmes que nous décrivons dans ce chapitre, le sens du parcours de ces zones n'est pas signifiant.

Algorithme 5.2 : ngbAlgoIB, skeleton algorithmique de travail sur le voisinage qui divise le traitement en deux parties, traitement dans la zone du bord et dans la zone de l'intérieur

```
1  ngbAlgoIB    ::    Streamize α → ExtrNgb α
2                  → Streamize α → ExtrNgb α
3                  → NgbOp α → Ar (I,I) α → Ar (I,I) α
4  ngbAlgoIB strmI extrI strmB extrB op ar = array (bounds ar)
5            $((     (zip ixsB)
6                  ∘(map op)
7                  ∘(map (extrB ar))
8                  $ixsB )
9             ++(
10                 (zip ixsI)
11                ∘(map op)
12                ∘(map (extrI ar))
13                $ixsI )
14            )
15       where   ixsB = strmB ar;  ixsI = strmI ar
```

La division en deux parties concerne également les fonctions d'extraction du voisinage qui sont du type ExtrNgb α dans la signature du type de la fonction ngbAlgoIB. Ainsi, nous pouvons avoir deux fonctions d'extraction du voisinage, une spécialisée pour la zone de l'intérieur, $extrI$, la deuxième adaptée à l'extraction dans la zone du bord, $extrB$. En revanche, le kernel de voisinage op reste le même pour les deux manières de traitement et la construction de l'array de sortie est semblable à celle de l'approche naïve. Le graphe de flux présenté sur la figure 5.5 démontre graphiquement la structure du skeleton algorithmique ngbAlgoIB sur des blocs fonctionnels et leur interconnexions.

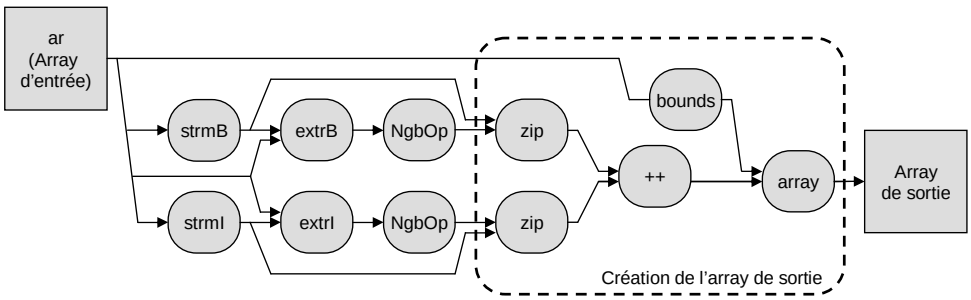


FIG. 5.5 : Graphe de flux exprimant le fonctionnement du skeleton algorithmique ngbAlgoIB

5.1.2.1 *Algorithmes concrets utilisant le skeleton algorithmique* ngbAlgoIB

Nous démontrons l'utilisation pratique du skeleton algorithmique ngbAlgoIB sur les mêmes exemples que ceux présentés pour l'approche naïve.

La dilatation par un élément structurant sur la grille carrée avec la valeur de bord constante en utilisant la division du traitement en traitement de l'intérieur et en traitement du bord est présentée par la fonction dilIBSQR :

```
dilIBSQR                :: (Ord α) ⇒ α → Ngb → Ar (I,I) α → Ar (I,I) α
dilIBSQR brdval ngb ar =
        ngbAlgoIB (streamI bb) (extrINgbSQR ngb)
                  (streamB bb) (extrBNgbSQR brdfnc ngb) ngbDilate ar
    where bb = ngbBB ngb;   brdfnc = cBorder brdval
```

Nous pouvons constater que nous utilisons une double approche au traitement. Les fonctions de parcours de l'image sont maintenant distinctes et sont devenues complexes car elles utilisent la fonction ngbBB pour déterminer le bounding box de l'élément structurant mais ce sont également les fonctions d'extraction de voisinage qui sont maintenant distinctes.

Nous définissons de la même manière la dilatation morphologique pour la grille hexagonale et le 6-voisinage avec la valeur de bord constante et travaillons en divisant le traitement en traitement de l'intérieur et du bord. Elle est présentée par la fonction dilIBHEXR :

```
dilIBHEXR               :: (Ord α) ⇒ α → Ngb → Ar (I,I) α → Ar (I,I) α
dilIBHEXR brdval ngb ar =
        ngbAlgoIB (streamI bb) (extrINgbHEXR ngb)
                  (streamB bb) (extrBNgbHEXR brdfnc ngb) ngbDilate ar
    where bb = ngbBB ngb;   brdfnc = cBorder brdval
```

Suivant la même logique, nous pouvons définir toutes les autres opérations morphologiques de base.

5.1.3 Généralisation du travail sur le voisinage

La spécialisation du traitement de l'image lors du travail avec les éléments structurants peut aller encore plus loin que à la division en deux zones comme présentée dans la section précédente 5.1.2.

Nous pouvons, en effet, choisir autant de zones particulières qu'il nous convient pour notre traitement. Ces zones vont correspondre à la spécialisation de traitement pour certains index dans l'image pour lesquelles les configurations de la position d'éléments structurants et des bords sont particulières. Dans notre logique, cette spécialisation essaie de minimiser les besoins d'échantillonnage inutile au-delà de l'image et nous fournit les procédures adaptées à ce but. La figure 5.6 nous montre un exemple d'une possible fragmentation du domaine de l'array.

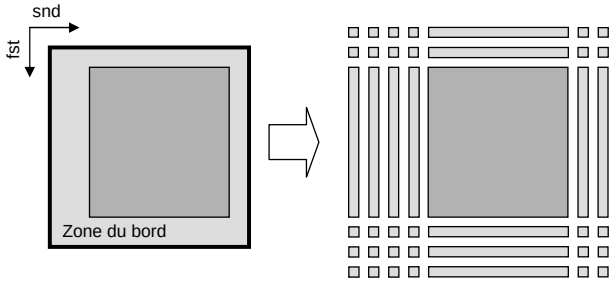
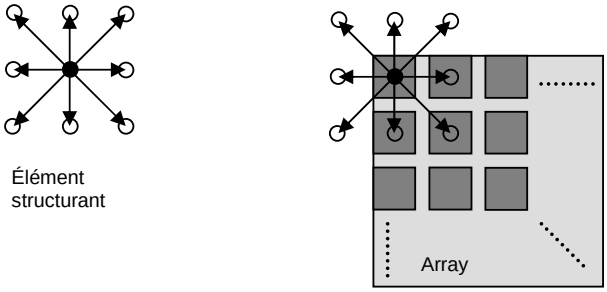


FIG. 5.6 : Décomposition du traitement de l'image à plusieurs zones dans lesquelles les traitements distincts peuvent être effectués

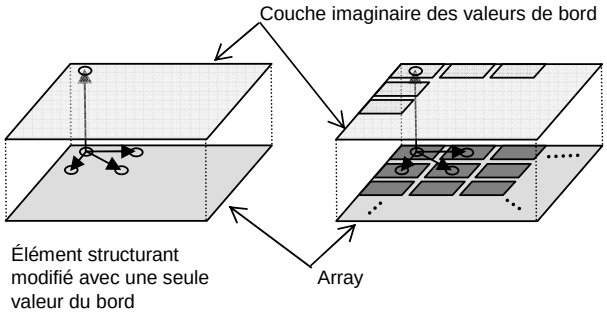
De plus, pour certaines opérations morphologiques, nous pouvons modifier non seulement le parcours et la manière d'extraire des voisins, mais également l'opération sur le voisinage sans changer l'exactitude du résultat final. Un bon exemple de cette situation est la dilatation locale du pixel placé dans le coin de l'image par l'élément structurant dont plusieurs vecteurs de déplacement pointent à l'extérieur

du domaine de l'image, cf. fig. 5.7(a). Si la valeur de bord est constante et indépendante de la position de l'élément structurant et si l'opération arithmétique ou logique de base que nous nous apprêtons à appliquer par la suite sur le voisinage local est idempotente envers les valeurs de bord, e.g. $max(x, x) = x$, il semble inutile d'appliquer l'échantillonnage de l'extérieur de l'image plus qu'une fois. Ainsi, nous pouvons épargner le temps du calcul lors de l'échantillonnage du voisinage mais également lors du calcul de l'opération morphologique locale sur ce dernier.

La fig. 5.7(b) nous montre, sur un exemple concret de la grille carrée et de 8-voisinage, la manière dont nous pouvons modéliser ce travail. Cette image est mise en contraste par rapport à la manière classique d'échantillonnage du voisinage local pour les pixels touchant le bord de l'image qui est présentée sur la fig. 5.7(a). Lors du travail avec l'élément structurant modifié, nous créons une couche imaginaire des valeurs de bord qui est représentée sur la fig. 5.7(b) superposée à l'image. Une seule valeur de bord est délivrée lors de l'évaluation des pixels touchant les bords de l'array. Ainsi, la fonction d'échantillonnage ne nous retourne que 5 valeurs du voisinage local à la place de 9 lors du travail classique et nous obtenons un stream moins large.



(a) Manière standard d'appliquer élément structurant



(b) Application d'un élément structurant modifié qui assume une seule valeur de bord par pixel touchant le bord

FIG. 5.7 : Exemple de la modification de l'élément structurant lors du traitement d'un pixel du coin de l'image convenable aux dilatations / érosions où une seule valeur de bord est assumée.

Dans le cas général, la fonction de traitement de voisinage local qui travaille avec ce stream doit être adaptée pour un tel traitement ce qui se traduit par le besoin de pouvoir spécifier non seulement la fonction de parcours de l'image et la fonction d'échantillonnage mais également la fonction travaillant sur le voisinage local pour des zones distinctes dans l'image. Ce qui nous conduit à la définition d'un nouveau skeleton algorithmique qui nous permettra de traiter spécifiquement chacune des zones sur lesquelles nous avons fractionné notre image.

5.1.3.1 Skeleton algorithmique généralisé de travail sur le voisinage ngbAlgoGen

Le skeleton algorithmique généralisé de travail sur le voisinage nous permet d'exprimer d'une façon formelle la possibilité de diviser le traitement le plus général possible, que nous avons présenté dans

5.1.1, page 99, en traitements plus spécifiques ou même en certain nombre des traitements entièrement adaptés à un cas particulier. Ce skeleton est représenté par la fonction ngbAlgoGen dans l'algorithme 5.3.

Algorithme 5.3 : ngbAlgoGen, skeleton algorithmique généralisé de travail sur le voisinage

```
1  ngbAlgoGen  ::  [Streamize α] → [ExtrNgb α] → [NgbOp α] → Ar (I,I) α → Ar (I,I) α
2  ngbAlgoGen strms extrs ops ar = array (bounds ar) (compute strms extrs ops ar)
3      where
4          compute [] [] [] ar = []
5          compute (s:ss) (e:es) (o:os) ar =
6                      (      (zip (s ar))
7                          ∘ (map o)
8                          ∘ (map (e ar))
9                          $ (s ar)
10                     )
11                 ++(compute ss es os ar)
```

Son fonctionnement exact suit le principe de fonctionnement du skeleton ngbAlgoIB (cf. page 104) qui divisait le traitement en deux parties. Le skeleton que nous présentons à cette place élargit les possibilités de traitement spécifique et pour assurer cela, il travaille avec les listes des fonctions de parcours de l'image, d'extraction du voisinage et des kernels assurant l'opération sur le voisinage.

La division de l'image en zones est assurée par l'ensemble des fonctions de parcours de l'image *strms* qui sont du type [Streamize α] et qui désignent ces zones par un stream des index. Cette division a la notion de segmentation avec laquelle nous travaillons couramment en morphologique mathématique lors de traitement du contenu des images. Ce qui est fractionné par la segmentation, c'est le domaine de l'image, c'est-à-dire l'ensemble des index désignant les pixels dans l'image. Le critère de la segmentation que nous utilisons ici est l'uniformité du calcul à effectuer. Donc, nous cherchons des zones où nous pouvons effectuer le traitement par la même fonction d'extraction du voisinage et le même kernel de réduction qui assurerait l'opération morphologique à l'échelle locale. Ainsi, nous posons les restrictions sur le fonctionnement de ces fonctions de parcours : nous exigeons une intersection vide entre les zones et nous exigeons que l'union de ces zones compose le domaine complet de l'image.

Les fonctions d'extraction du voisinage qui sont appliquées spécifiquement dans chacune des zones sont indiquées par la liste *extrs* qui est du type [ExtrNgb α] et les fonctions de l'opération sur le voisinage sont exprimées par la liste *ops* qui est du type [NgbOp α]. L'array d'entrée est spécifié par l'argument *ar*.

Le corps de ce skeleton utilise, sur les lignes 6 à 9, une expression que nous connaissons déjà du skeleton de l'approche naïve, cf. l'algorithme 5.1, page 100, et de l'approche qui divisait le domaine de l'image en deux zones, cf. l'algorithme 5.2, page 104. Mais il l'encapsule ici dans la fonction interne compute qui traite les trois streams des fonctions d'une façon récursive et pré-compose le stream résultant des tuples (index, valeur) par la fonction de la composition des stream ++ (ligne 11). Ce stream des résultats partiels sert à la fin du traitement à la composition de l'array de sortie par la fonction **array** sur la ligne 2. Notons que l'expression sur la ligne 4 :

```
compute [] [] [] ar = []
```

désigne la fin de la récursion de la fonction compute pour le cas spécial des arguments où les trois streams d'entrée sont vides.

La figure 5.8 nous montre l'interprétation graphique du fonctionnement de ce skeleton.

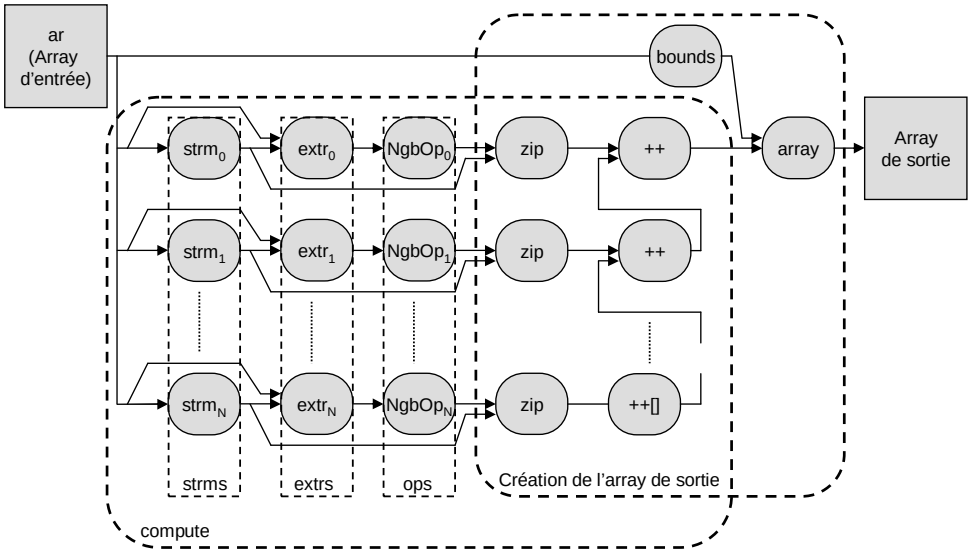


FIG. 5.8 : Graphe de flux exprimant le fonctionnement du skeleton algorithmique ngbAlgoGen de traitement général de voisinage

5.1.4 Approche des superpixels, algorithmes aux kernels complexes qui exploitent la localité des données

Le travail *pixel par pixel* que nous avons utilisé jusqu'à présent dans nos skeletons algorithmiques et qui sera utilisé aussi dans tous les algorithmes spécifiques de la morphologie mathématique dérivés de ces skeletons, est seulement une des possibilités. Il s'agit du travail qui suit quasi-directement les définitions de base des opérations morphologiques et même s'il est facilement transposable aux architectures massivement parallèles, il est loin d'être le plus intéressant du point de vue de la structure d'un algorithme et, surtout, du point de vue des performances.

Une catégorie des algorithmes, fréquemment utilisés dans la pratique, emploie les éléments structurants particuliers qui ont la forme d'un disque (ou d'une boule) unité. Dans l'espace digitalisé, nous employons les éléments structurants qui approximent la boule unité et qui, pour la grille donnée, désignent les voisins d'un pixel. Ces algorithmes sont largement utilisés dans les algorithmes géodésiques. Cette catégorie des algorithmes morphologiques utilise les données proches d'un pixel dans l'espace 2D et sont, lors du traitement, stockées dans la mémoire cache la plus rapide ou présents dans les registres. Ainsi, à un instant du traitement, nous disposons d'un accès aux données le plus rapide qui puisse être.

La localité spatiale d'un groupe de pixels qui se partagent mutuellement une partie de leurs voisins est un phénomène qui n'est surtout pas à ignorer. Il est donc évident que nous avons la volonté d'exploiter au maximum la localité de ces données et, sur certaines architectures, également leur présence dans les registres lors du traitement. Ainsi, il serait beaucoup plus intéressant de travailler avec un certain groupe de pixels à la fois plutôt que de les traiter en employant la méthode *pixel par pixel*. Il serait intéressant lors d'un tel traitement de pouvoir réutiliser certaines des valeurs des voisins d'un pixel qui ont été déjà extraites de l'image et peut-être même de réutiliser les résultats partiels des opérations du voisinage.

Le phénomène d'extraction des valeurs des pixels voisins, plus précisément des pixels qui sont désignés par l'élément structurant, entre également en jeu. L'extraction du voisinage se modifie dans ce cas et nous procédons, en effet, à l'extraction du voisinage de ce groupe de pixels. Nous pouvons percevoir ce travail également comme le travail sur le voisinage local, même si la notion "*local*" concerne maintenant tout un groupe de pixels et le terme "*local élargi*" serait plus approprié.

Si nous percevons l'extraction de tels groupes de pixels comme le traitement en stream avec un kernel d'extraction, le calcul morphologique sur ce voisinage *local élargi* peut également être perçu comme le calcul par un kernel de traitement. Dans ce cas, le kernel sera adapté au traitement de tout un groupe de

pixels et son fonctionnement sera décrit par une fonction plus complexe que celles que l'on a pu voir lors de traitement d'un voisinage d'un seul pixel (e.g. fonction ngbDilate, page 92).

Puisque cette manière de travailler suit la logique du traitement *pixel par pixel* décrit précédemment, nous avons décidé d'employer dans ces algorithmes le concept des *superpixel*s dont le traitement aura une structure semblable à celle d'un seul pixel. Rappelons que le concept des superpixels est formellement introduit dans la section 4.4.5, page 73, du chapitre 4 qui est consacré au formalisme fonctionnel adopté pour la morphologie mathématique.

Il faut noter que le traitement d'un superpixel est un traitement spécialisé et en tant que tel, il exige une implémentation particulière pour un élément structurant donné et, par conséquent, il demande beaucoup plus d'effort lors du développement pour sa mise en œuvre. Pour poursuivre l'explication des algorithmes pour les superpixels, nous allons, tout d'abord, définir un skeleton algorithmique qui nous formalisera la structure des algorithmes de voisinage qui sont non-dépendants du sens du parcours et qui travaillent avec les superpixels.

5.1.4.1 Skeleton algorithmique généralisé pour le traitement morphologique par superpixels

La structure des algorithmes travaillant sur le voisinage par l'approche des superpixels va combiner deux approches, les deux étant déjà mentionnées. Nous nous baserons sur le skeleton généralisé de travail sur le voisinage ngbAlgoGen qui travaillait avec les éléments de base d'un array. Nous allons le transposer au travail généralisé avec des superpixels, tout en suivant les principes de passage à flux de superpixels (et vice versa) et les principes d'extraction du voisinage des superpixels que nous avons décrits dans la section 4.4.5, page 73, dédiée à cette problématique.

Ainsi, nous créons un nouveau skeleton algorithmique que nous appelons ngbAlgoGenSP et qui est présenté par l'algorithme 5.4.

Algorithme 5.4 : ngbAlgoGenSP, skeleton algorithmique généralisé de travail sur le voisinage pour les superpixels

```
1  ngbAlgoGenSP  ::  [StreamizeSP α] → [ExtrNgbSP α] → [NgbOpSP α] → [ZipSP α]
2                    → Ar (I,I) α → Ar (I,I) α
3  ngbAlgoGenSP strms extrs ops zips ar = array (bounds ar) (compute strms extrs ops zips ar)
4      where
5          compute [] [] [] [] ar = []
6          compute (s:ss) (e:es) (o:os) (z:zs) ar =
7                    (    (foldl1 (++) )
8                       ∘ (map z)
9                       ∘ (zip (s ar))
10                      ∘ (map o)
11                      ∘ (map (e ar))
12                      $ (s ar)
13                   )
14                  ++(compute ss es os zs ar)
```

Cet algorithme prend quatre listes des fonctions qui spécifient le traitement exact des superpixels dans certaines zones de l'image. La philosophie de découpage de l'image en zones où différentes approches du traitement peuvent être appliquées est la même que pour les éléments de base d'une image, cf.5.1.3.1, page 106.

La première liste, *strms* contient les fonctions de passage d'un array à un stream des index d'ancrage. La deuxième liste, *extrs*, contient des fonctions d'extraction du voisinage d'un superpixel à partir de son index d'ancrage ; la troisième, *ops* applique un kernel du calcul sur ce voisinage élargi et la quatrième liste nous fournit les fonctions qui assurent le passage d'un superpixel aux éléments de base indexés par

leurs position dans l'image et stockés dans un tuple (index, valeur). Le cinquième paramètre d'entrée est l'array ar.

De toutes ces fonctions, ce sont effectivement celles qui décrivent le kernel du calcul qui sont les plus intéressantes. C'est pourquoi nous allons nous concentrer sur la description de ces dernières dans la suite de ce chapitre et nous allons y présenter 3 cas précis pour le traitement à l'intérieur d'un superpixel : pour la grille carré de 4 et 8 voisins et pour la grille hexagonale de 6 voisins. Ces cas sont spécialisés pour les opérations morphologiques dont la fonction du traitement du voisinage local d'un pixel est une fonction de réduction par une seule opération de base[1]. Donc, nos opérations cibles sont la dilatation morphologique (réduction par la fonction max) et la fonction de l'érosion morphologique (réduction par la fonction min).

Ces 3 cas ne sont, en effet, que 3 des cas possibles et très concrets de l'implémentation. D'autres implémentations sont envisageables, par exemple avec des superpixels plus larges, mais nous croyons que celles que nous présentons ici suffisent à démontrer le principe de fonctionnement et une brève analyse des performances.

5.1.4.2 Traitement d'un superpixel sur la grille carrée et 8-voisins pour la dilatation / l'érosion

Le kernel de traitement que nous présentons ici est spécialisé pour les dilatations et les érosions sur la grille carrée et le voisinage constitué du pixel central et de ses 8 voisins. Il s'agit, en effet, d'un kernel qui est le plus facile à expliquer car la manière dont les données sont réutilisées d'un élément à l'autre est facile à interpréter visuellement.

Expliquons la fonction de ce kernel sur le graphe de flux, q.v. fig. 5.9. Cette figure présente, dans sa partie gauche, les éléments constituant le voisinage d'un superpixel. Les éléments correspondant au superpixel y sont entourés par une ligne intermittente. Le réseau d'interconnexions qui lie les éléments de l'image avec les blocs opérationnels *op*2 et *op*3 montre comment on procède pour calculer le superpixel résultant (présenté également entouré par une ligne intermittente). Le bloc opérationnel *op*2(,) désigne une fonction de base prenant deux arguments, le bloc *op*3(, ,) désigne une fonction ayant 3 arguments.

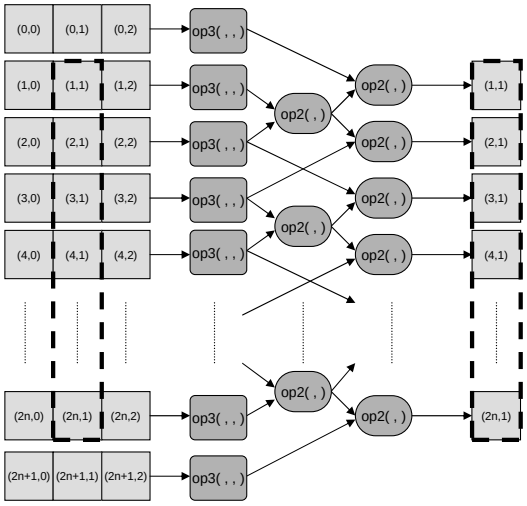


FIG. 5.9 : Fonctionnement du kernel traitant un superpixel en 8-voisinage

Lors du travail avec un superpixel, nous pouvons profiter au maximum de la localité des données, de leur présence dans la mémoire la plus rapide de la hiérarchie des caches mais également des résultats partiels d'évaluation d'un élément que nous réutilisons lors du calcul. Tel que présenté sur la fig. 5.9, le kernel du calcul travaille avec un superpixel ayant un nombre pair des éléments de base ($2n$) dans

[1] Les kernels du calcul qui ne sont pas de ce type, par exemple le kernel pour la transformation *tout ou rien*, seraient construits différemment.

la première dimension. Cette contrainte est posée par ce cas particulier mais nous pouvons envisager la construction d'un kernel du calcul qui travaillerait avec un nombre impair des éléments, même si d'un point de vue pratique nous préférons travailler avec des superpixels qui fragmentent l'image d'une manière simple et qui sont, par conséquent, de dimensions multiples de 2.

5.1.4.3 Traitement d'un superpixel sur la grille carrée et 4-voisins pour la dilatation / l'érosion

La construction d'un kernel du calcul pour un superpixel qui utilise la grille carrée et le voisinage constitué d'un pixel central et de ses 4 voisins ne nous mène pas à une structure d'interconnexions des éléments et des blocs fonctionnels aussi compréhensible que l'on a pu le voir dans 5.1.4.2. C'est, en effet, dû au fait que le nombre de voisins qui sont partagés entre deux pixels classiques est limité à 2, c'est-à-dire que nous ne pouvons réutiliser qu'une seule opération de base.

La fig. 5.9 nous démontre le graphe de flux expliquant le fonctionnement de ce traitement. Notons, comme on l'a fait pour le superpixel travaillant avec 8-voisins, que le superpixel doit avoir un nombre pair des éléments de base ($2n$) pour ce traitement précis. Le traitement d'un superpixel d'un nombre impair des éléments est envisageable, même si d'un point de vue pratique nous préférons travailler avec des superpixels dont les dimensions sont divisibles par 2 pour les mêmes raisons que l'on a présentées pour le traitement de 8-voisins.

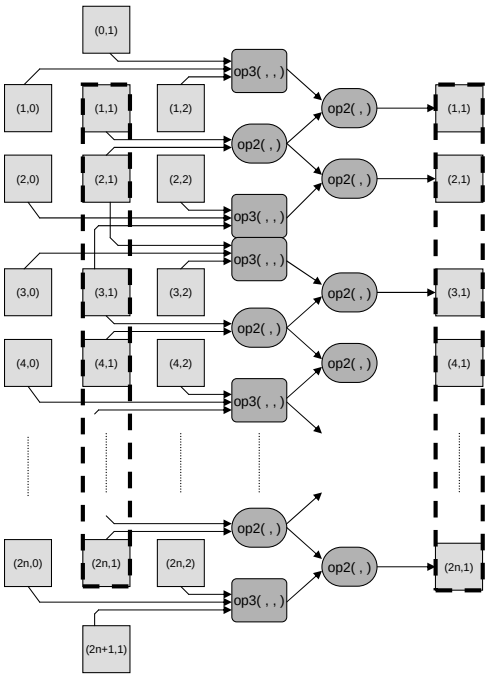


FIG. 5.10 : Fonctionnement du kernel traitant un superpixel en 4-voisinage

5.1.4.4 Traitement d'un superpixel sur la grille hexagonale décalée par lignes et 6-voisins pour la dilatation / l'érosion

Le traitement sur la grille hexagonale est assez particulier, dû au décalage des lignes lors du traitement. Pour que l'exemple de fonctionnement que nous présentons sur la fig. 5.11 soit cohérent avec la définition mathématique de la grille hexagonale (cf. 4.6, page 84), le superpixel doit être constitué d'un nombre pair des pixels ($2n$) et l'élément désigné par l'index d'ancrage, l'index (1,1) sur cette figure, doit être placé sur une ligne impaire. La construction d'un kernel du calcul pour la grille hexagonale avec le décalage par colonnes est facilement dérivable à partir du principe de travail que nous présentons sur cette figure.

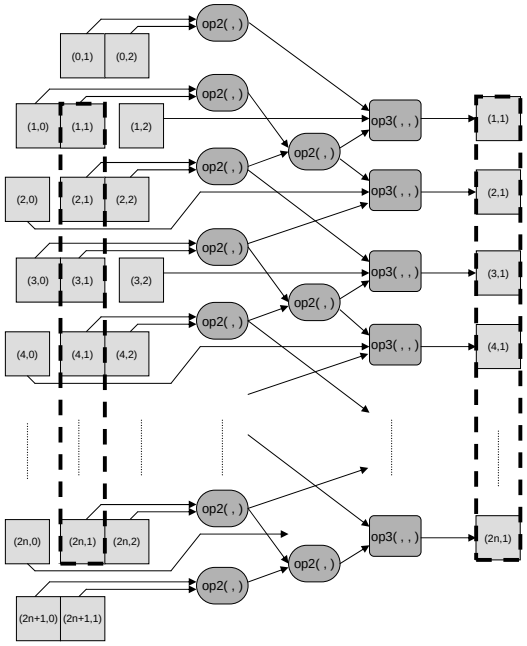


FIG. 5.11 : Fonctionnement du kernel traitant un superpixel en 6-voisinage

5.2 Algorithmes élémentaires pour les GPPMM

Le traitement des images par la morphologie mathématique sur les architectures multimédia possédant des capacités SWAR est, pour les algorithmes non dépendants du sens de parcours de l'image, directement dérivable du style de travail que l'on vient de présenter pour les GPP sans les fonctionnalités multimédia. Nous allons, en effet, exploiter au maximum les possibilités du calcul vectoriel. Ce travail est particulier et se traduit :

- par l'utilisation des blocs de données, exprimés dans le formalisme fonctionnel par le type du vecteur paqueté PVec.
- par le travail particulier que nous allons effectuer lors d'accès aux valeurs voisines d'un vecteur paqueté dans la mémoire. En général, l'emplacement des voisins d'un vecteur paqueté dans la mémoire ne coïncide pas avec la perception d'un array comme array de vecteurs paquetés. Les voisins d'un vecteur paqueté, qui sont désignés par les déplacements relatifs en unités de types de base de l'array, peuvent, dans le cas général, être stockés dans la mémoire à l'emplacement qui est partagé par deux vecteurs paquetés voisins. Ce fait nous conduira à l'utilisation des instructions de la lecture non-alignée, c'est à dire de la lecture des données vectorielles qui ne sont pas stockées dans la mémoire sur les adresses alignées (adresses qui sont divisibles sans résidu par la taille d'un vecteur paqueté).
- par l'utilisation des fonctions SIMD, arithmétiques ou logiques, lors du traitement des valeurs composant le voisinage ou les valeurs désignées par la liste des vecteurs de déplacement dans le cas général.

5.2.1 Skeletons algorithmiques GPPMM de base

La structure de fonctionnement des algorithmes morphologiques de base pour les architectures SWAR est, en effet, la même que pour les architectures générales. C'est à cette place que nous allons faire un parallèle entre les algorithmes pour les GPPMM et les algorithmes pour les GPP et c'est à cette place où nous allons voir en pratique la force de la généralisation qui nous est fournie par les skeletons algorithmiques définis en formalisme fonctionnel.

En effet, les skeletons algorithmiques que nous avons présentés pour le travail avec les GPP sont

autant généraux qu'ils englobent également le travail avec les types de vecteurs paquetés. Ceci dit, les skeletons algorithmiques pour les GPPMM ne sont que des spécialisations des skeletons pour un type de données particulier – le type polymorphe de vecteurs paquetés (PVec I α).

L'approche naïve à l'implémentation des algorithmes morphologiques dans l'esprit de travail SIMD est définie par le skeleton algorithmique ngbAlgoSIMD présenté par l'algorithme 5.5. C'est la signature de type qui est importante dans cette définition car elle nous précise qu'il s'agit de la fonction qui est une spécialisation du skeleton général ngbAlgo de l'approche naïve qui partage avec elle le corps.

Algorithme 5.5 : ngbAlgoSIMD, skeleton algorithmique de l'approche naïve pour le travail SIMD sur le voisinage

```
1 ngbAlgoSIMD :: Streamize (PVec I α) → ExtrNgb (PVec I α) → NgbOp (PVec I α)
2                → Ar (I,I) (PVec I α) → Ar (I,I) (PVec I α)
3 ngbAlgoSIMD = ngbAlgo
```

Nous appliquons le même principe de spécialisation également sur le skeleton ngbAlgoIB pour en ensuite obtenir un nouveau skeleton ngbAlgoIBSIMD, défini par l'algorithme 5.6, qui utilise des procédés distincts dans la zone intérieure et dans la zone du bord de l'image et qui est spécifique au traitement SIMD.

Algorithme 5.6 : ngbAlgoIBSIMD, skeleton algorithmique de travail SIMD sur le voisinage qui divise le traitement en deux parties, traitement dans la zone du bord et dans la zone de l'intérieur

```
1 ngbAlgoIBSIMD :: Streamize (PVec I α) → ExtrNgb (PVec I α)
2                  → Streamize (PVec I α) → ExtrNgb (PVec I α)
3                  → NgbOp (PVec I α) → Ar (I,I) (PVec I α) → Ar (I,I) (PVec I α)
4 ngbAlgoIBSIMD = ngbAlgoIB
```

Dans la cas d'une fragmentation générale du domaine de l'image lors du traitement morphologique SIMD, nous obtenons le skeleton algorithmique ngbAlgoGenSIMD qui est également une spécialisation d'un skeleton défini auparavant. Il s'agit de la spécialisation du skeleton ngbAlgoGen (cf. l'algorithme 5.3, page 5.3).

Algorithme 5.7 : ngbAlgoGenSIMD, skeleton algorithmique généralisé de travail SIMD sur le voisinage

```
1 ngbAlgoGenSIMD :: [Streamize (PVec I α)] → [ExtrNgb (PVec I α)] → [NgbOp (PVec I α)]
2                   → Ar (I,I) (PVec I α) → Ar (I,I) (PVec I α)
3 ngbAlgoGenSIMD = ngbAlgoGen
```

5.2.2 Algorithmes concrets GPPMM de base de la morphologie mathématique

Les algorithmes de base de la morphologie mathématique concrets pour le traitement SIMD sur les architectures multimédia se bâtissent à partir des skeletons présentés dans la section précédente 5.2.1. Ils emploieront les fonctions qui spécialiseront l'usage de ces skeletons pour le travail avec les données paquetées en se basant sur les primitives fonctionnelles que nous avons définies pour ce type de données dans la première partie de cette thèse, q.v. chapitre 4. Notamment, il s'agit des primitives du parcours de l'image, des primitives de l'extraction du voisinage et des primitives d'opération sur le voisinage pour la morphologie mathématique.

Présentons alors un exemple concret. Le premier que nous décrivons est l'algorithme de la dilatation morphologique sur la grille carrée par l'élément structurant ngbSQR4. La fonction dilSQRSIMD décrit un algorithme qui utilise la fonction générale d'extraction du voisinage pour la grille carrée et pour l'échantillonnage général (sampGen) qui travaille avec la fonction de traitement du bord cBorder rendant une valeur constante pour tous les éléments du bord.

```
dilSQRSIMD :: (Ord α) ⇒ (PVec I α) → Ngb → Ar (I,I) (PVec I α) → Ar (I,I) (PVec I α)
dilSQRSIMD brdval ngbs ar = ngbAlgoSIMD
                              indices
                              (extrNgbSQRSIMD how n (sampGen (cBorder brdval)) ngbs)
                              ngbDilate
                              ar
  where  n = rangeSize∘bounds$(ar!lo); (lo,hi) = bounds ar
```

D'autres fonctions peuvent être construites en utilisant le schéma présenté. En se basant sur un autre skeleton algorithmique, en choisissant la fonction d'extraction du voisinage, d'échantillonnage ou de traitement des bords appropriée et en choisissant la fonction du kernel de l'opération morphologique, nous pouvons dériver les implémentations des opérations morphologiques de base adaptées à un cas particulier d'utilisation, notamment les opérations sur les grilles hexagonales et la méthode plus complexe de gestion des bords (i.e. par réflexion des bords).

5.2.3 Algorithmes SIMD basés sur l'approche des superpixels

Notre idée de travail avec les vecteurs paquetés peut également être transposée à l'approche des superpixels. Dans ce cas, les superpixels ne seront pas constitués des éléments de base mais des vecteurs paquetés.

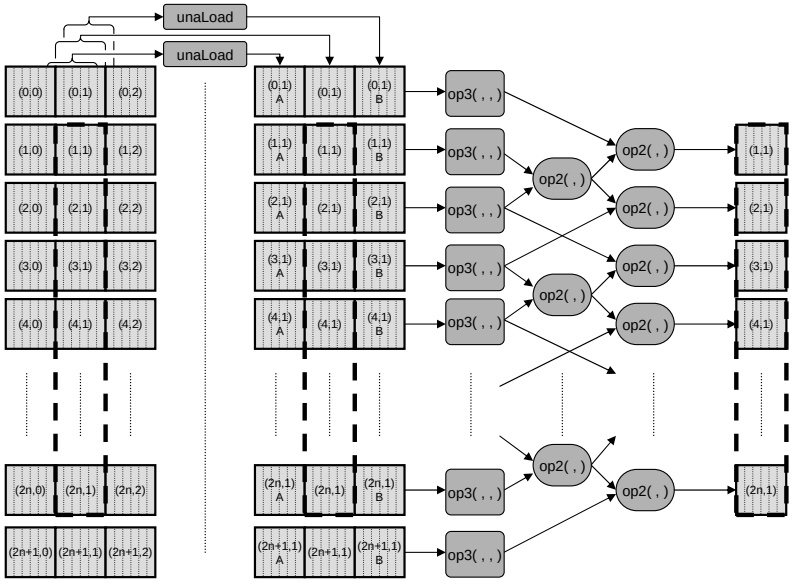


FIG. 5.12 : Fonctionnement du kernel traitant un superpixel SIMD en 8-voisinage

Si nous étudions la transposition de cette idée plus en détail, nous nous apercevons que nous avons recours, lors de l'extraction des voisins d'un superpixel, au même phénomène que celui que l'on a pu rencontrer lors de l'extraction des voisins d'un vecteur paqueté et qui nous a mené à l'utilisation des lectures non-alignées des données vectorielles de la mémoire.

La figure 5.12 illustre cette situation sur le traitement SIMD d'un superpixel pour l'élément structurant définissant le voisinage avec 8 voisins sur la grille carrée. Elle diffère, par rapport à la figure 5.9 qui assure la même opération morphologique sur les GPP, dans l'utilisation des vecteurs paquetés à la place des éléments de base et surtout dans la présence des opérations de lecture non alignée, représentée par

les blocs *unaLoad*. Notons que pour ce cas particulier où nous connaissons a priori la définition exacte de l'élément structurant, nous n'utilisons pas la lecture non alignée pour l'extraction des vecteurs paquetés centraux mais nous l'utilisons uniquement pour l'extraction des voisins de ces derniers. Ces voisins sont dénotés dans l'image par les index A et B.

D'autres schémas de travail avec les superpixels SIMD peuvent être dérivés, notamment ceux pour le travail avec le 4 pixels voisins sur la grille carrée, cf. sa version scalaire sur la fig. 5.10, page 111, et pour le travail avec 6 voisins sur la grille hexagonale, cf. sa version scalaire sur la fig. 5.11, page 112.

Algorithme 5.8 : ngbAlgoGenSPSIMD, skeleton algorithmique généralisé de travail sur le voisinage pour les superpixels

```
1 ngbAlgoGenSPSIMD :: [StreamizeSP(PVecI α)] → [ExtrNgbSP(PVec I α)]
2                   → [NgbOpSP(PVec I α)] → [ZipSP(PVec I α)]
3                   → Ar (I,I) (PVec I α) → Ar (I,I) (PVec I α)
4 ngbAlgoGenSPSIMD = ngbAlgoGenSP
```

La formalisation de l'approche des superpixels SIMD travaillant avec les types des vecteurs paquetés suit la même logique que celle employée pour les algorithmes élémentaires. Nous reprenons le skeleton algorithmique ngbAlgoGenSP (défini par l'algortihme 5.4) et nous spécialisons sa forme générique pour le type polymorphe de vecteurs paquetés PVec I α en tant que fonction ngbAlgoGenSPSIMD, définie par l'algorithme 5.8.

5.3 Algorithmes géodésiques pour les GPP/GPPMM

5.3.1 Idée de base

Le traitement géodésique constitue un type de traitement de base de la morphologie mathématique. On désigne une opération comme opération morphologique géodésique si le traitement est effectué d'une manière itérative, si nous pouvons y percevoir une propagation des valeurs et si cette propagation est restreinte lors d'enchaînement des itérations par une fonction, exprimée le plus souvent en tant qu'image du *masque*. Citons comme exemple[Ser00] la définition de la dilatation δ géodésique de taille unité (donné par l'élément structurant B) de la fonction g restreinte par la fonction f :

$$\delta_f(g) = \inf(g + B, f) \tag{5.1}$$

nous travaillons à l'échelle des fonctions, c'est-à-dire à l'échelle des images entières. Si on suivait ces définitions, la restriction, représentée dans notre exemple par la fonction inf, devrait opérer sur les images.

Pour les raisons de performance, il est préférable d'imposer la restriction sur la propagation déjà dans le kernel du calcul de l'opération sur le voisinage qui travail à l'échelle des éléments de l'array. Ainsi, nous pouvons construire un kernel d'une fonction morphologique géodésique à condition que nous puissions introduire la valeur restrictive (valeur du masque correspondant au pixel traité) comme un paramètre supplémentaire dans la fonction définissant le kernel.

Nous allons nous appuyer sur les kernels géodésiques des opérations sur le voisinage. Nous avons présenté la forme formelle pour ces kernels dans la section 4.6.6, page 94. Elle sera représentée dans nos algorithmes par les fonctions du type NgbGOp. De plus, nous devons également gérer dans ces algorithmes l'extraction de la valeur du masque. Pour cette extraction, nous allons nous appuyer sur la même fonction du parcours des stream que celle utilisée pour l'image même. Mais il nous faut spécifier une fonction d'échantillonnage des éléments d'un array à partir d'un index qui est du type SampFnc et qui sera utilisée pour obtenir la valeur concrète du masque pour un index concret.

Remarquons que pour extraire correctement les éléments qui se correpondent dans l'image et dans le masque, les dimensions de ces deux arrays doivent être obligatoirement les mêmes. L'algorithme 5.9

nous décrit la définition du skeleton algorithmique ngbGAlgoGen destiné pour le travail géodésique sur le voisinage.

Algorithme 5.9 : ngbGAlgoGen Skeleton algorithmique généralisé de travail géodésique sur le voisinage

```
1  ngbGAlgoGen  ::   [Streamize α] → ([ExtrNgb α],[SampFnc α])→ [NgbGOp α]
2                    → (Ar (1,1) α, Ar (1,1) α) → Ar (1,1) α
3  ngbGAlgoGen strms (extrsN,extrsM) ops (ar, msk) = array
4                                                    (bounds ar)
5                                                    (compute strms (extrsN,extrsM) ops ar msk)
6      where
7        compute [] [] [] ar msk = []
8        compute (s:ss) ((eN:esN),(eM,esM)) (o:os) ar msk =
9                  (    (zip (s ar))
10                    $(map2 o
11                        ((map (eN ar )) $ (s ar))
12                        ((map (eM msk)) $ (s msk))
13                      )
14                 )
15                ++(compute ss (esN,esM) os ar msk)
```

La fig. 5.13 nous illustre sur le diagramme du flux la structure et la fonction de ce skeleton dans le cas spécial où nous ne fractionnons pas l'image aux différentes zones et où nous avons, par conséquent, son fonctionnement décrit par une seule fonction de parcours de l'image, une seule fonction d'extraction des voisins et une seule fonction de kernel opérant sur le voisinage et sur le masque (*strms*, *extsN*, *extsP*, *ops* sont les listes contenant un seul élément).

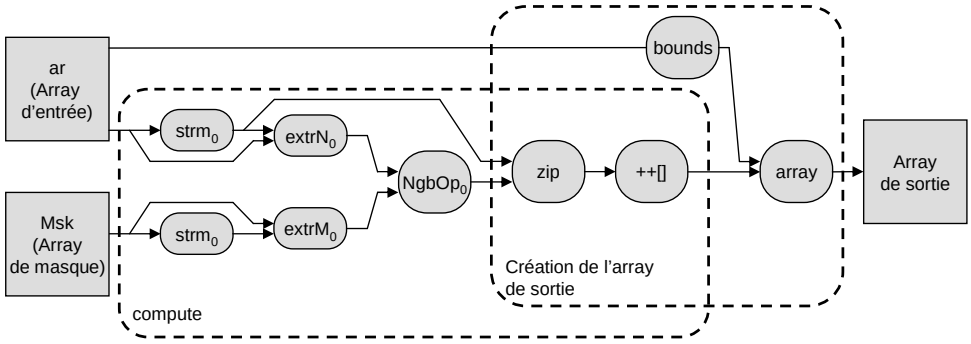


FIG. 5.13 : Fonctionnement du skeleton algorithmique ngbGAlgoGen de traitement du voisinage avec le masque dans le cas spécial où nous ne fractionnons pas l'image aux différentes zones

5.3.2 Itérations, fin de propagation

Ce skeleton est élémentaire et il ne décrit qu'une itération de la propagation géodésique. Pour l'utiliser correctement, l'élément structurant employé doit obligatoirement être de la taille unité. La propagation géodésique elle-même sera obtenue par l'incorporation de ce skeleton dans un skeleton ou un algorithme gérant la propagation. Il existe, en effet, deux types de fonctions géodésiques :

- fonctions géodésiques avec le nombre d'itérations donné,
- fonctions géodésiques itérant jusqu'à l'idempotence.

Pour les premières, le nombre d'itérations est connu d'avance est la construction de l'algorithme peut s'appuyer sur ce fait. Pour les deuxièmes, nous devons tester l'arrêt de la propagation (test d'idempotence de la fonction géodésique sur une image donnée). Ce test compare les valeurs de l'image d'entrée (avant

l'emploi de l'opération géodésique) et l'image de sortie (après avoir effectué cette opération géodésique). Si elles sont les mêmes, nous arrêtons la propagation car nous avons atteint l'état d'idempotence.

Ce test est assez coûteux en terme d'opérations arithmétiques car il doit effectuer une comparaison pixel par pixel et employer ensuite une fonction de réduction qui, pour l'image toute entière, ne fournit qu'une seule valeur reflétant l'identité / non-identité des deux images. Cette valeur peut être binaire ou pas. En termes de flux de données, nous effectuons d'abord l'opération d'application d'une fonction (comparaison) sur tous les éléments du stream qui est suivie par la réduction de ce stream à une seule valeur.

Pour diminuer l'impact de ce test de la fin de la propagation, plusieurs approches sont à prévoir pour son implémentation, dépendantes surtout du caractère de notre image. Nous pouvons tester la propagation à chaque itération, ce qui peut s'avérer coûteux. Mais nous pouvons envisager d'utiliser la propriété d'idempotence qui est atteinte à la fin de la propagation. Ainsi, nous savons a priori que chaque itération supplémentaire ne changera pas les résultats. En se basant sur cette propriété, nous pouvons construire un algorithme qui ne teste pas la fin de la propagation après chaque itération mais qu'après un certain nombre d'itérations.

5.3.3 Note sur le travail géodésique avec les superpixels

L'approche des superpixels est également utilisable pour les algorithmes géodésiques de la morphologie mathématique avec certaines modification de la structure des skeletons algorithmiques. En effet, la fonction du parcours de l'image pour les superpixels (qui est du type StreamizeSP) nous retourne le stream des index d'ancrage des superpixels.

Les valeurs des index d'ancrage issues de ce stream seront utilisées par les fonctions qui extraient le voisinage large d'un superpixels (fonctions du type ExtrNgbSP) mais également par les fonction d'échantillonnage d'un superpixels (fonctions du type SampFncSP) qui nous retourne une liste des valeurs d'un superpixel.

La construction du skeleton algorithmique ngbGAlgoGenSP pour le travail géodésique sur le superpixels, présenté par l'algorithme 5.10, s'appuie sur le skeleton algorithmique de base (ngbGAlgoGen) dont elle utilise le corps. Il s'agit, en effet de la spécialisation de cette fonction pour le travail avec le superpixels, comme nous pouvons le voir sur la signature de type dans l'algorithme 5.10.

Algorithme 5.10 : ngbGAlgoGen Skeleton algorithmique généralisé de travail géodésique sur le voisinage par l'approche des superpixles

```
1 ngbGAlgoGenSP :: [StreamizeSP α] → ([ExtrNgbSP α],[SampFncSP α]) → [NgbGOpSP α]
2                  → (Ar (I,I) α, Ar (I,I) α) → Ar (I,I) α
3 ngbGAlgoGenSP = ngbGAlgoGen
```

5.3.4 Travail SIMD avec les vecteurs paquetés

Les opérations géodésiques pour les vecteurs paquetés sont directement dérivables en tant que spécialisation de fonctionnement du skeleton géodésique ngbGAlgoGen (présenté par l'algorithme 5.9) car ce dernier est assez général pour englober également le travail avec les vecteurs paquetés. L'algorithme 5.11, qui définit le skeleton algorithmique ngbGAlgoGenSIMD, nous présente la forme formelle de cette spécialisation.

Le travail SIMD pour les superpixels est également possible. La manière dont on construirait un skeleton algorithmique qui travaillerait avec les superpixels composés des vecteurs paquetés et avec des superpixels de l'image du masque est intelligible à partir de la description que nous avons faite dans cette section.

Algorithme 5.11 : ngbGAlgoGenSIMD Skeleton algorithmique généralisé de travail géodésique SIMD sur le voisinage

```
1 ngbGAlgoGenSIMD :: [Streamize (PVec I α)] → [ExtrNgb (PVec I α)] → [NgbGOp (PVec I α)]
2                    → (Ar (I,I) (PVec I α), Ar (I,I) (PVec I α))
3                    → Ar (I,I) (PVec I α)
4 ngbGAlgoGenSIMD = ngbGAlgoGen
```

5.4 Algorithmes pour les GPU

Cette section sera consacrée aux algorithmes morphologiques de base, non dépendants du sens du parcours de l'image, qui utilisent les moyens fonctionnels du pipeline graphique et qui sont destinés à être exécutés sur les processeurs graphiques.

Il est possible d'envisager plusieurs approches à la construction de ces algorithmes. Nous mentionnons :

- L'approche utilisant les opérations de Minkowski, q.v. page 118.
- L'approche utilisant l'échantillonnage de textures dans l'unité de traitement des fragments, q.v. page 120.
- L'approche utilisant les *point sprites*, q.v. page 120.

5.4.1 Traitement des bords de la texture sur les GPU

Notre explication des algorithmes pour les GPU commencera par l'explication des capacités matérielles particulières des processeurs graphiques pour l'échantillonnage des textures avec les index pointant en dehors du domaine de la texture. Il s'agit des capacités de la gestion des bords lors de l'échantillonnage de la texture. Dans les algorithmes utilisés couramment de la morphologie mathématique, nous allons faire appel le plus souvent à la fonction de la valeur constante du bord de l'image mais d'autres possibilités (les valeurs reflétées etc.) peuvent être envisagées.

Sachant que l'image à traiter est représentée dans le GPU en tant que texture, nous pouvons bénéficier de ces capacités pour simplifier la construction des algorithmes morphologiques travaillant sur le voisinage dans ce matériel. Ainsi, le calcul à effectuer sera le même pour tout le domaine de l'image, y compris pour les zones dans lesquelles l'élément structurant désigne les pixels à l'extérieur de ce dernier.

L'uniformité du calcul est un phénomène important dans ce cas, car nous n'aurons pas à traiter plusieurs zones de l'image avec une fonction du kernel spécifique pour chacune des zones. Cette uniformité va se traduire par l'utilisation de la commande géométrique couvrant entièrement notre domaine d'intérêt – rectangle de dimension de l'image dans ce cas précis.

Remarquons que les fonctionnalités de la gestion des bords sont ajustées lors de la création de la texture et ne sont effectuées qu'une fois dans la phase de l'initialisation de notre algorithme.

5.4.2 Approche utilisant les opérations de Minkowski

Une possible approche pour implémenter les algorithmes de la morphologie mathématique utilise les opérations de Minkowski (cf. la section 4.6.3 traitant d'éléments structurants, page 86). Il s'agit de l'approche qui est décrite par Hopf et Ertl dans leur article[HE00], le plus ancien (année 2000) que nous ayons pu consulter et qui implémentait les opérations morphologiques sur les GPU utilisant les capacités matérielles qui étaient à disposition à l'époque.

Sachant que les GPU grand public gardent la compatibilité rétroactive, il est possible d'envisager l'utilisation de cette approche également sur les processeurs graphiques modernes, mais son utilité pratique est moindre car ces nouveaux processeurs nous offrent d'autres possibilités pour une implémentation plus rapide. En revanche, les méthodes qui sont basées directement sur les opérations de Minkowski

peuvent trouver leur emploi sur les architectures des processeurs graphiques dédiés (e.g. à basse consommation) qui n'offrent que les fonctionnalités restreintes par rapport aux processeurs graphiques grand public.

Le principe de cette approche suit la méthode de travail des opérations de Minkowski, connues comme les opérations de base dans la théorie de la morphologie mathématique. Lors du calcul de ces opérations, nous déplaçons l'image entière selon les vecteurs définissant les éléments structurants et nous fusionnons les valeurs placées sur la même position avec une opération logique (et/ou) dans le cas de traitement binaire ou avec une opération arithmétique (min/max) dans le cas de traitement en niveaux de gris.

Dans l'implémentation sur les GPU, l'image d'entrée est exprimée en tant que texture. Lors du calcul, nous travaillons avec les commandes graphiques en forme de rectangles qui ont les dimensions de l'image. Avant d'envoyer la commande graphique dans le GPU, la position des vertex est modifiée selon les vecteurs de déplacement de l'élément structurant. Ainsi, nous obtenons une séquence des commandes contenant les rectangles avec les positions décalées dans l'espace mais qui contiennent les mêmes coordonnées pour l'indexation de la texture. De cette manière, nous définissons les emplacements distincts dans le framebuffer sur lesquells nous allons effectuer le rendu des pixels à partir de la texture. La figure 5.14 illustre cette situation.

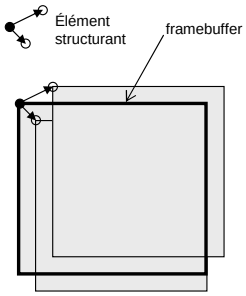


FIG. 5.14 : Utilisation des opérations de Minkowski sur les GPU pour le calcul morphologique. Les rectangles à rendre sont décalés selon les vecteurs de l'élément structurant

Dans ce cas précis, nous n'employons pas un programme particulier pour le traitement des vertex et des fragments et nous programmons le pipeline de telle manière qu'il n'effectue qu'une opération d'échantillonnage pour chaque fragment créé. L'opération morphologique choisie (dilatation ou érosion) est effectuée par les opérations du blending (max pour la dilatation, min pour l'érosion) dans la phase de post-traitement des fragments lors de leur fusion avec les valeurs déjà présentes dans le framebuffer. Donc, le kernel de réduction qui réduit le stream de valeurs de voisinage à une seule valeur résultante intervient dans le bloc des raster-opérations avec le framebuffer présenté dans notre modèle formel par le raster processeur (`rprocessor`).

Généralement, le framebuffer doit être préparé à ce traitement et doit être rempli par une valeur constante qui a un comportement neutre vis-à-vis de notre opération morphologique (valeur minimale du type de stockage pour la dilatation, valeur maximale du type de stockage pour l'érosion) dans la phase de l'initialisation de l'algorithme. Dans le cas où notre élément structurant contient également le vecteur (0,0) désignant le pixel central, nous pouvons pré-remplir le framebuffer par l'image même en copiant les données de la texture (qui sont celles de l'image) dans le framebuffer[1]. Dans ce cas, la liste des commandes graphiques contiendra un rectangle de moins, correspondant au déplacement par le vecteur (0,0), et nous allons, en effet, travailler avec la liste des vecteurs désignant les voisins plutôt que avec l'élément structurant entier (qui contient le vecteur désignant le pixel central).

Les résultats sont obtenus dans le framebuffer après que le rendu du dernier rectangle est effectué. Ils peuvent être récupérés pour le calcul suivant dans le GPP via le transfert des données à partir du

[1] Notons qu'il n'est pas possible d'utiliser une zone de la mémoire du GPU pour la lecture et l'écriture en même temps. L'opération de copie est alors nécessaire dans ce cas.

framebuffer. Ou ils peuvent également être réutilisés lors de la prochaine application de l'opération de la morphologique mathématique. Nous profiterons partiellement du fait que les données sont déjà présentes dans le framebuffer et l'initialisent ainsi dans le cas où l'élément structurant de l'opération suivante contiendrait le vecteur désignant le pixel central. Nonobstant que le framebuffer soit ainsi initialisé, nous devons procéder à une copie de ses valeurs vers une texture qui serait utilisée pour l'échantillonnage dans l'opération suivante.

5.4.3 Approche utilisant l'échantillonnage complexe des textures dans l'unité de traitement des fragments

Les processeurs graphiques plus récents offrent la possibilité d'exécuter le programme sur chacun des fragments. Nous allons exploiter cette capacité matérielle pour pouvoir introduire une autre approche à l'implémentation des opérations morphologiques de base sur les processeurs graphiques.

L'idée de cette approche consiste à utiliser l'échantillonnage massif de la texture, effectué dans l'unité de traitement des fragments lors de l'application du kernel de traitement sur le stream des fragments. La forme précise de cet échantillonnage est dictée par la forme de l'élément structurant, qui, une fois traduit à une liste des vecteurs de déplacement dans le domaine des index, peut être utilisée pour accéder aux valeurs des éléments de la texture. Remarquons que l'opération morphologique est calculée également dans l'unité des fragments. Par conséquent, le fragment modifié résultat porte l'information sur le résultat de cette opération morphologique comme la valeur de la couleur qui est à inscrire dans le framebuffer.

Dans les algorithmes de ce type, notre commande graphique ne sera composée que d'un seul rectangle qui couvre la surface dans le framebuffer correspondant au domaine de l'image et sur laquelle nous affichons les résultats du traitement. Notons également que nous travaillons avec une seule image de texture. Nous travaillerons avec plusieurs images de texture dans le cas où nous voudrions effectuer une opération morphologique plus complexe faisant appel à plusieurs images, e.g. les opérations géodésiques en font partie.

L'initialisation du framebuffer n'est pas nécessaire dans ce cas car l'opération morphologique de réduction des valeurs du voisinage est calculée entièrement par le processeur des fragments (fprocessor). Ainsi, nous n'avons pas besoin de faire appel aux opérations du *blending* dans la phase de post-traitement des fragments. En effet, nous n'avons pas besoin des opérations raster pour effectuer le kernel de l'opération morphologique de base, mais nous pouvons utiliser la fonctionnalité du *blending* pour distribuer le calcul et alléger la charge des unités de traitement des fragments, notamment lors du travail avec les opérations géodésiques qui utilisent un masque à niveaux de gris. Si le masque est binaire, l'application du masque par le *blending* ne semble pas la meilleure solution car les GPU possèdent la fonctionnalité du *stencil buffer*, également englobé dans le bloc des raster opérations lors du post-traitement des fragments, qui assure exactement cette opération.

Remarquons que l'exécution de ce kernel de traitement des fragments est parallélisée dans les processeurs graphiques par le paradigme de la réplication fonctionnelle (cf. le skeleton farm, page 67). Sachant que c'est le bloc de traitement des fragments qui est le plus parallélisé au niveau du matériel, cette approche à l'implémentation a toutes les prédispositions pour profiter pleinement des capacités des GPU modernes. La figure 5.15 illustre le fonctionnement de cette approche. Nous y démontrons, pour un exemple concret d'un élément structurant, de quelle manière on procède lors de l'extraction des texels (des données à partir de la texture) désignés par cet élément structurant.

5.4.4 Approche utilisant les *point sprites*

Parmi les capacités particulières des GPU pour le calcul destiné à la synthèse des images, nous trouvons une fonctionnalité intéressante qui peut nous servir, effectivement, pour implémenter les opérations morphologiques sur les GPU – les *point sprites*.

Il s'agit d'une technique spéciale, mais couramment utilisée dans la synthèse d'images pour créer facilement les objets de 2 dimensions et les placer dans l'image. Le principe de cette technique est simple.

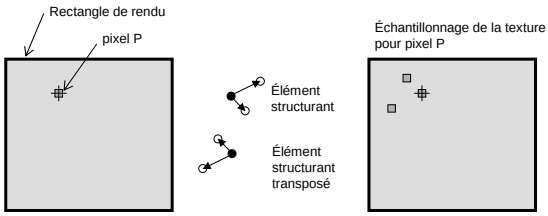


FIG. 5.15 : Utilisation d'échantillonnage de la texture dans l'unité de traitement des fragment pour l'extraction du voisinage

Nous voudrions afficher dans le framebuffer un objet rectangulaire et texturé à partir d'un minimum de commandes. Donc, notre primitive géométrique qui passera par la commande graphique du GPP au GPU est un point. C'est d'ici que l'on a dérivé leur appellation *point sprites*.

En effet, c'est l'unité d'assemblage des primitives qui, étant préalablement configurée pour ce travail, remplace chaque point désignant un *point sprite* par un rectangle dont les vertex contiennent les index pointant vers la texture associée à ce *point sprite* et dont la position est relative à la position du point qui a été passé par la commande graphique.

En effet, nous voulions exploiter cette technique pour une autre version de l'implémentation des opérations de Minkowski. Supposant que nous puissions créer un *point sprite* ayant les dimensions de notre image à traiter, nous pourrions percevoir une séquence des *point sprites* comme la version graphique de l'élément structurant. En effet, chaque point de cette liste définirait le déplacement de l'image et si nous avions bien configuré l'unité du blending, de la même manière que nous l'avons fait dans 5.4.2, page 118, pour l'approche utilisant les opérations de Minkowski, nous aurions obtenu les résultats de notre opération morphologique de base dans le framebuffer.

Le problème de cette méthode est qu'il n'est pas habituel de traiter les *point sprites* qui sont de dimensions égales à celles de l'image et que le support de grandes dimensions lors de ce travail varie d'un GPU à l'autre. En effet, les GPU d'ATI devrait avoir cette possibilité tant que notre GPU d'expérimentation, NVidia GeForce 6800, permettrait d'afficher les *point sprites* de seulement 63×63 pixels.

Ainsi, nous laissons cette technique comme problème ouvert pour l'avenir. Notons que même si cette méthode présente de grandes similitudes avec celle qui utilisait les opérations de Minkowski (cf. 5.4.2, page 118), elle permet de passer moins d'informations par une commande graphique ; si cette dernière idée ne participait pas à l'augmentation signifiante du temps d'exécution (car on n'épargne qu'un nombre très peu important de données lors du transfert de la commande graphique par le bus liant le GPP avec le GPU), les *point sprites* pourraient se présenter comme une méthode idéale à l'écriture plus logique des programmes morphologiques pour les GPU car la signification et la manière de fonctionner des *point sprites* sont très proches des vecteurs définissant les éléments structurants.

5.4.5 Description des algorithmes pour les processeurs graphiques par le formalisme fonctionnel

Dans cette section, nous voulons démontrer comment nous construisons des algorithmes pour les GPU et comment nous pouvons décrire un tel algorithme formellement utilisant les définitions mentionnées dans la section 4.5, page 76, du chapitre 4 qui présentait le modèle formel du traitement en pipeline graphique.

Nous présentons d'abord l'algorithme 5.12, défini par la fonction gpuTexture, qui est trivial qui ne fait qu'échantillonner une texture et n'applique aucune opération supplémentaire. Donc, cet algorithme effectue le rendu de la texture dans le framebuffer. Il prend deux paramètres, cmd est la commande graphique qui va décrire la zone pour le rendu et e est l'environnement du travail du GPU. Notons que cet environnement doit être initialisé pour ce travail ; il doit contenir une texture (spécifiée par son index txi) et le framebuffer du GPU doit être également prêt à accueillir des résultats (e.g. une zone de mémoire de texture connectée au framebuffer).

Algorithme 5.12 : gpuTexture, skeleton algorithmique d'affichage d'une texture dans le framebuffer du GPU

```
1  gpuTexture  ::  Commands → Env → Env
2  gpuTexture  cmd e = pipeGPU vpid rasterize fpTexture rpid cmd e
3
4  fpTexture  ::  FProg
5  fpTexture e (p, d, cs, [(txi,txp)]) = mkF p d [(1,col)] [(txi,txp)]
6      where
7          col      = smpBorder ( (getTXs $ e)!! txi )  txp
```

Le fonctionnement de cet algorithme s'appuie sur le bloc de traitement des fragments et, par conséquent, l'algorithme spécifie le fragment programme fpTexture pour cet affichage. Le reste du pipeline n'effectue que le passage d'une unité à l'autre – vertex programme vpid retourne le vertex inchangé, l'unité de rastérisation rasterize effectue la rastérisation standard et nous n'utilisons aucune fonctionnalité spécifique de post-traitement des fragments (exprimé par rpid).

Le fonctionnement du fragment programme se compose d'appel d'une seule fonction – fonction d'échantillonnage des textures smpBorder. Ses deux arguments sont la texture, spécifiée par l'expression

```
( (getTXs $ e)!! txi )
```

et qui récupère la texture de l'environnement, et par la position txp du texel à échantillonner.

L'algorithme de la dilatation suit la logique présentée. Nous modifions le fragment programme pour ce but. Plutôt qu'échantillonner un seul texel de la texture, nous procédons à l'échantillonnage de tout le voisinage spécifié par la liste des vecteurs de déplacement *ngb*. Dans le cœur de la fonction fpDilate définissant le fragment programme, nous créons (ligne 7) la fonction d'échantillonnage *smpfnc* par l'application partielle de la fonction smpBorder à la texture qui est récupérée de l'environnement *env* de la même manière que dans l'algorithme précédent. Nous appliquons, ligne 8, la fonction de spécialisation du voisinage specNgbSQR pour obtenir des index pointant vers les pixels du voisinage. Ensuite, l'échantillonnage par la fonction $smpfnc$ est appliquée et nous fournit les valeurs des pixels $samples$. La valeur résultante col est obtenue, ligne 9, par l'application de la fonction morphologique de dilatation, ngbDilate.

L'initialisation de l'environnement de travail du GPU doit être effectué de la même façon que décrit précédemment.

Algorithme 5.13 : gpuTexture, skeleton algorithmique de la dilatation, image est stockée dans la texture, résultat sera écrit dans le framebuffer du GPU

```
1  gpuDilate  ::  Ngb → Commands → Env → Env
2  gpuDilate  ngb cmd e = pipeGPU vpid rasterize (fpDilate $ ngb) rpid cmd e
3
4  fpDilate  ::  Ngb → FProg
5  fpDilate ngb e (p, d, cs, [(txi,txp)]) = mkF p d [(1,col)] [(txi,txp)]
6      where
7          smpfnc  = smpBorder ( (getTXs $ e)!! txi )
8          samples = map smpfnc (specNgbSQR ngb txp)
9          col     = ngbDilate samples
```

5.5 Résultats expérimentaux

Le sujet exposé dans ce chapitre est fondamental pour les implémentations des opérations de base de la morphologie mathématique. Vu que ces dernières sont abondamment utilisées dans les applications de traitement des images par la morphologie, il est primordial de pouvoir exécuter ces opérations le plus rapidement possible.

C'est pourquoi nous voulions explorer les performances des méthodes proposées dans ce chapitre et pourquoi nous présentons, dans la tab. 5.1, les résultats des tests comparatifs de l'opération de dilatation morphologique pour les méthodes que nous avons développées nous-mêmes (*GPU* et *MorphoMedia*) au cours de cette thèse et des méthodes assurant les mêmes fonctionnalités mais qui sont incorporées d'une manière standard dans les produits commerciaux (*Aphelion* et *MATLAB*) qui représentent ainsi la référence industrielle. Une autre implémentation que nous avons mise en relation avec nos résultats est celle qui utilise les méthodes de l'outil logiciel Morphée, développé au Centre de Morphologie Mathématique pour les besoins de la recherche algorithmique et qui peut être considéré comme référence d'une bibliothèque des fonctions spécialement dédié aux algorithmes morphologiques mais programmées d'une façon générique.

Notons que notre implémentation, désignée par *MorphoMedia*, utilise l'approche des superpixels SIMD, cf. 4.4.5, page 73, lors de l'évaluation avec les macro blocs ayant 64 pixels dans l'axe de stockage des données et la dimension de l'image dans la coordonnée perpendiculaire. L'implémentation de la dilatation sur les processeurs graphiques, désignée par *GPU*, utilise l'approche de l'échantillonnage complexe des textures dans les unités de traitement des fragments, cf. 5.4.3, page 120.

La fig. 5.16 présente graphiquement les données de la tab. 5.1. Nous constatons que pour une opération aussi basique que la dilatation, les temps de calcul pour les implémentations commerciales existantes sont excessivement longs si on les compare avec les temps des implémentations des algorithmes pour les GPP issus de cette thèse. Ces derniers offrent des taux d'accélération intéressants, allant jusqu'à 230 par rapport au logiciel MATLAB ; cela pour une image de 256 ko et le cas de l'élément structurant de 4 voisins sur la grille carrée ("*SQR DISC2D 4*") de taille 10.

Des temps encore plus intéressants sont ceux des processeurs graphiques. Nous constatons des taux d'accélération supérieurs à 2 par rapport à notre implémentation des superpixels SIMD, la meilleure obtenue sur le processeur général. Ce sont, en effet, ces résultats-là qui valorisent l'effort que nous avons investi dans l'exploitation de l'utilisation des processeurs graphiques pour le calcul morphologique. Ces résultats sont d'autant plus encourageants si nous mentionnons les caractéristiques de notre matériel. Tandis que le processeur GPP Intel Pentium 4 a été cadencé à 2.4 GHZ, la carte graphique NVidia GeForce 6800 LT ("*light edition*") est dotée d'un GPU qui appartient au bas de gamme parmi les processeurs offrant les fonctionnalités qui nous intéressent et a été cadencée à 375 MHz.

5.6 Récapitulation

Nous avons présenté, dans ce chapitre, la construction des algorithmes de la morphologie mathématique dont le point commun est l'indépendance du traitement sur le sens du parcours de l'image. Il s'agit, en effet, des algorithmes qui exposent le parallélisme des données et des tâches et sont ainsi les bons candidats à l'exécution en parallèle.

Nous avons exploré la piste de l'utilisation des moyens matériels qui sont à disposition dans les architectures multimédia - la parallélisation appartenant à la catégorie SIMD qui est effectuée à l'échelle des registres (où on parle également du traitement SWAR). Nous avons proposé la méthodologie pour la création des algorithmes traitant des données en tant que flux pour ce type d'architectures et nous avons démontré leur description dans le formalisme fonctionnel.

Une autre piste que nous avons explorée dans ce chapitre est celle de l'utilisation des processeurs graphiques pour le calcul massivement parallélisé des algorithmes de la morphologie. Vu que les processeurs graphiques sont les architectures particulières de traitement des données en tant que flux, il nous semblait approprié d'ajouter l'expérience avec ces processeurs dans ce chapitre.

SQR DISC2D 8 Taille = 1

Image	GPU	GPP			
		MrphMedia	Morphée	Aphelion*	MATLAB 7.0**
	ms	ms	ms	ms	ms
256 ko	—	0.372	50.0	86.0	84
1 Mo	1.1	2.587	171.9	223.5	331
4 Mo	—	36.376	631.2	609.4	1312

SQR DISC2D 4 Taille = 1

Image	GPU	GPP			
		MrphMedia	Morphée	Aphelion*	MATLAB 7.0**
	ms	ms	ms	ms	ms
256 ko	—	0.216	46.7	pas de fonction	200
1 Mo	0.65	1.956	156.9	pas de fonction	858
4 Mo	—	32.876	586.0	pas de fonction	3046

SQR DISC2D 8 Taille = 10

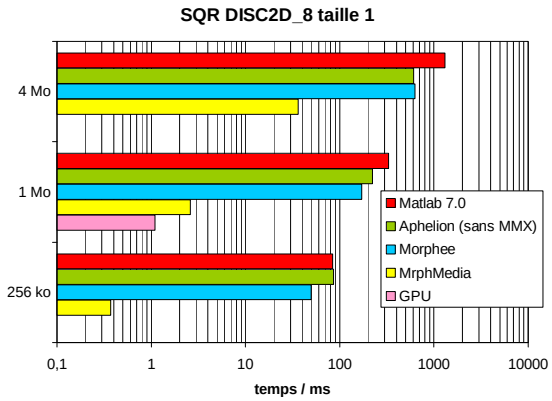
Image	GPU	GPP			
		MrphMedia	Morphée	Aphelion*	MATLAB 7.0**
	ms	ms	ms	ms	ms
256 ko	—	3.769	460.6	118.7	75
1 Mo	—	21.081	1574.0	346.9	260
4 Mo	—	386.876	5821.9	1117.2	929

SQR DISC2D 4 Taille = 10

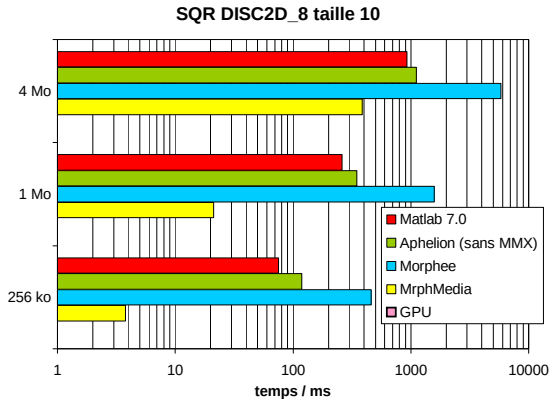
Image	GPU	GPP			
		MrphMedia	Morphée	Aphelion*	MATLAB 7.0**
	ms	ms	ms	ms	ms
256 ko	—	2.281	423.8	pas de fonction	527
1 Mo	6.6	13.544	1463.8	pas de fonction	1727
4 Mo	—	352.252	5465.7	pas de fonction	6216

* Fonctions d'Aphelion non-MMX ; ** MATLAB utilise la décomposition d'élément structurant SQR DISC2D 8 aux éléments structurant segments lors de ce calcul. Dimensions des images : 256 ko = 256x256x4x8bit, 1 Mo = 512x512x4x8bit, 4 Mo = 1024x1024x4x8bit. GPP = Intel Pentium 4 à 2.4 GHz (single thread, 8 ko L1, 512 ko L2) ; GPU = NVidia GeForce 6800 LT AGP 4x à 375 MHz

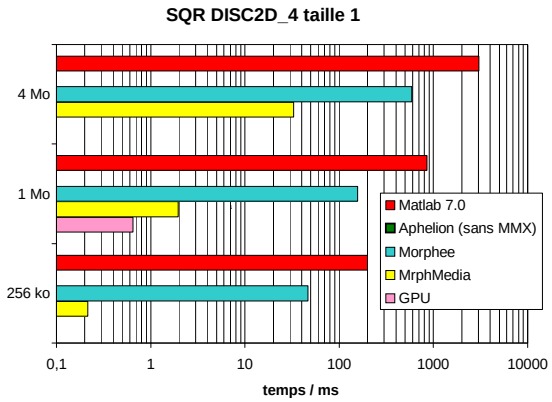
TAB. 5.1 : Résultats expérimentaux de diverses implémentations de la dilatation morphologique



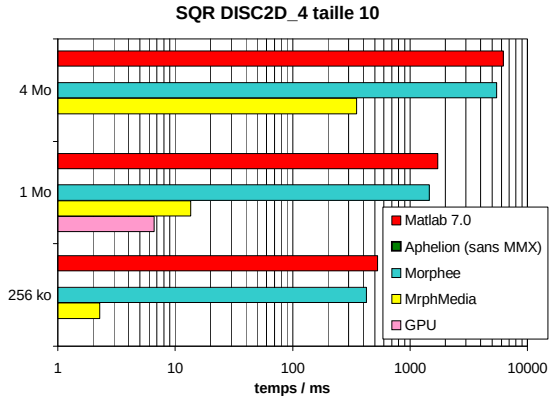
(a) grille carrée, élément structurant de 8 voisins, taille 1



(b) grille carrée, élément structurant de 8 voisins, taille 10



(c) grille carrée, élément structurant de 4 voisins, taille 1



(d) grille carrée, élément structurant de 4 voisins, taille 10

FIG. 5.16 : Comparaison des temps d'exécution de la dilatation pour différents logiciels et différentes implémentations, sur les GPP/GPPMM (Intel Pentium 4 à 2,4 GHz ; 512 ko L2 ; classe 0-F-2-4-1E) et les GPU (NVidia GeForce FX 6800 ; AGP 4x à 375 MHz). Les temps pour les GPU n'incluent pas les transferts.

Ce fait nous a permis également de comparer leurs performances avec les implémentions correspondantes exécutées sur le processeur général lors de l'étude comparative de l'exécution de l'opération de base de la morphologie – la dilatation. Les résultats ont non seulement prouvé la supériorité de notre approche des superpixels SIMD pour l'exécution des opérations morphologiques sur les GPP par rapport aux produits commerciaux existants, mais ont également démontré que l'implémentation de la même opération sur les GPU est encore plus rapide (avec le taux d'accélération supérieur à 2) par rapport à l'approche des superpixels SIMD qui assurait la meilleure exécution sur les GPP.

Cette page est blanche par intention

CHAPITRE 6

Permutation SIMD des arrays appliquée au changement de stockage des données vectorielles

Les processeurs du calcul général présentent, en ce qui concerne le traitement d'images, certains désavantages. L'un de ceux-ci, propre à toutes les architectures du type von-Neumann, est sans doute la façon linéaire d'organisation et d'adressage des données dans la mémoire de travail. L'espace de mémoire est perçu par le processeur comme unidimensionnel car il est présenté comme tel par le sous-système de gestion de la mémoire

Les données composées qui présentent l'aspect régulier d'une grille et qui possèdent 2 ou plusieurs dimensions, i.e. arrays, matrices, images, tenseurs, sont stockées dans la mémoire linéaire et l'accès à ces données est assuré par une fonction de mapping des index. Ainsi, chaque index d'une structure ayant 2 ou plusieurs coordonnées est mis en correspondance avec un index dans la mémoire linéaire. Cette organisation linéaire de la mémoire est souvent à l'origine des problèmes considérables dans le domaine du traitement d'images, particulièrement ressenti lors de l'utilisation de l'approche vectorielle SIMD où l'on a besoin d'accéder à plusieurs éléments de l'image à la fois par le biais des données paquetées.

C'est pourquoi nous constatons un besoin essentiel de réorganiser les données vectorielles. Pour cela, nous introduisons les algorithmes qui changent mutuellement l'axe principal de stockage avec l'axe secondaire, i.e. l'axe x pour y et vice versa. Nous n'insistons pas pour autant sur l'utilisation de la même fonction d'indexation des données. En effet, notre besoin prioritaire est de pouvoir accéder aux données par les instructions vectorielles SIMD et nous pouvons nous contenter d'une autre manière d'indexation car celle-ci joue un rôle secondaire.

Les opérations qui satisfont notre demande pour le changement mutuel de l'axe principal avec l'axe secondaire sont :

- transposition par diagonale,
- transposition par antidiagonale,
- rotation de $+\frac{\pi}{2}$,
- rotation de $-\frac{\pi}{2}$.

Dans la section suivante 6.1 intitulée *Transpositions et rotations des arrays*, nous allons présenter les définitions de ces opérations. Elles nous seront utiles d'un côté pour donner des définitions formelles d'opérations peu courantes dans le calcul matriciel (notamment les rotations des arrays), d'un autre côté, elles vont constituer automatiquement les descriptions des algorithmes triviaux qui travaillent élément par élément. Dans la section suivante, 6.2, nous présenterons l'approche qui utilise les macro blocs pour l'évalutation de toutes ces opérations. Cette approche nous servira plus tard, dans la section 6.3, page 133, intitulée *Algorithmes rapides SIMD de transposition et de rotation*, où nous allons présenter l'approche SIMD aux transpositions et aux rotations des arrays pour les architectures SWAR et qui va utiliser les macro blocs pour son travail.

6.1 Transpositions et rotations des arrays

La première catégorie des algorithmes convenant à la réorganisation des données pour l'échange mutuel des axes est constituée par les algorithmes de transposition d'un array par la diagonale et l'anti-diagonale. La deuxième catégorie est constituée par les algorithmes de rotation d'un array de $+\frac{\pi}{2}$ de $-\frac{\pi}{2}$.

Dans la pratique, nous allons nous servir plus souvent des transpositions, surtout pour leur propriété d'*auto-inversion*, où le même algorithme appliqué sur un array X deux fois de suite donne comme résultat l'array d'origine X. Pour les transpositions par la diagonale, T_D, et les transpositions par l'anti-diagonale, T_A, les expressions suivantes sont valides :

$$X = T_D(T_D(X))$$
$$X = T_A(T_A(X))$$

Ce qui n'est pas le cas pour les rotations de $+\frac{\pi}{2}$, R_+, et de rotation de $-\frac{\pi}{2}$, R_-, qui assurent mutuellement les opérations inverses. Cette propriété est décrite par les expressions suivantes :

$$X = R_-(R_+(X))$$
$$X = R_+(R_-(X))$$

Même si en pratique nous n'utiliserons que les transpositions, nous ajoutons à notre explication également les rotations car elles sont connexes au sujet de changement de la manière de stocker et, comme nous le verrons par la suite, entrent entièrement dans la logique de la construction des algorithmes suivants.

6.1.1 Définitions des transpositions et des rotations

6.1.1.1 Définition de la transposition par diagonale

La définition de la *transposition par la diagonale* (principale) d'un array de 2 dimensions, comme nous le définissons par la fonction tr2DDiag, est identique à la définition d'une transposition matricielle.

```
tr2DDiag     ::  Ar (I,I) α → Ar (I,I) α
tr2DDiag ar = array ((1,1), (q,p)) [((i,j), ar!(j,i)) | i ← [1..q], j ← [1..p]]
    where (p,q) = dimsAr2D ar
```

Dans cette définition, nous utilisons la fonction *array* qui crée un nouvel array à partir de l'étendue des index et d'un liste des tuples (index, élement). La définition est simple, pour un élément avec l'index (i, j) dans l'array de sortie nous associons un élement avec l'index (j, i) de l'array d'entrée.

6.1.1.2 Définition de la transposition par antidiagonale

La *transposition par l'antidiagonale*, définie par la fonction tr2DADiag, est similaire à la définition précédente à l'exception du travail avec les index :

```
tr2DADiag     ::  Ar (I,I) α → Ar (I,I) α
tr2DADiag ar = array ((1,1), (q,p))
                     [((i,j), ar!(p−j+1, q−i+1)) | i ← [1..q], j ← [1..p] ]
    where (p,q) = dimsAr2D ar
```

6.1.1.3 Définition de la rotation de $+\frac{\pi}{2}$

Nous définissons la *rotation de* $+\frac{\pi}{2}$ par la fonction rot2DPlus90 de la même manière en appliquant un travail différent sur les index :

```
rot2DPlus90     ::  Ar (I,I) α → Ar (I,I) α
rot2DPlus90 ar = array ((1,1), (q,p))
                       [((i,j), ar!(j,q−i+1)) | i ← [1..q], j ← [1..p] ]
    where (p,q) = dimsAr2D ar
```

6.1.1.4 Définition de la rotation de $-\frac{\pi}{2}$

Pareillement, la *rotation de* $-\frac{\pi}{2}$ d'un array est défini par la fonction rot2DMinus90 :

```
rot2DMinus90    ::  Ar (I,I) α → Ar (I,I) α
rot2DMinus90 ar = array ((1,1), (q,p))
                        [((i,j), ar!(p−j+1, i)) | i ← [1..q], j ← [1..p]]
    where (p,q) = dimsAr2D ar
```

6.1.1.5 Définitions de la fonction commune pour les transpositions et les rotations

Nous définissons l'algorithme 6.1, un algorithme trivial pour le changement du sens de stockage, qui travaille élément par élément et selon les définitions de ces quatre opérations. Cet algorithme définit la fonction commune pour ces quatre opérations, trRot2D. Nous utiliserons avantageusement cette fonction qui choisit l'opération exacte selon une clé dans nos prochains algorithmes :

Algorithme 6.1 : trRot2D, définit la fonction commune des transpositions par diagonale et par antidiagonale et des rotations de $+\frac{\pi}{2}$ et de $-\frac{\pi}{2}$ travaillant élément par élément directement selon les définitions de ces opérations

```
1  trRot2D    ::  [Char] → Ar (I,I) α → Ar (I,I) α
2  trRot2D how ar | how == "TD"  = tr2DDiag      ar
3                 | how == "TA"  = tr2DADiag     ar
4                 | how == "R+"  = rot2DPlus90   ar
5                 | how == "R−"  = rot2DMinus90  ar
```

6.2 Approche macro blocs aux transpositions et rotations

Bien que nous puissions utiliser les transpositions des arrays selon leurs définitions travaillant élément par élément, notre intérêt pour les architectures à flux nous incite à exploiter le parallélisme de données, à trouver les algorithmes qui seraient plus intéressants pour le traitement en parallèle et, par conséquent, plus efficaces que si on utilisait le traitement séquentiel.

Le coeur de notre approche se situe dans le travail par blocs. Nous allons découper un array d'une façon régulière et prédéfinie à des *macro blocs*. La figure 6.1 illustre cette situation pour le découpage d'un array à 3×3 macro blocs.

Il est à remarquer qu'en théorie, les dimensions d'un macro bloc peuvent ne pas être égales et de plus, elles ne sont pas obligées d'être ni de puissance 2, ni un multiple de 2. La transposition d'un bloc $M \times N$ d'une dimension impaire est possible, en se réduisant, pour les dimensions égales à 1×1, à un cas trivial de découpage par élément. En pratique, nous allons utiliser plutôt les blocs convenables à notre architecture matérielle, le plus souvent des dimensions de multiples de 2 et de puissance 2. Mais il est possible d'envisager, pour les architectures dédiées, une solution différente, e.g. des dimensions de multiples de 2 mais pas de puissance 2.

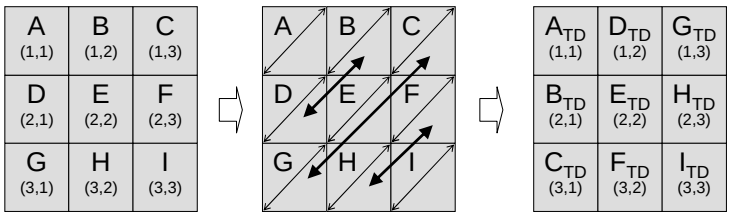
A (1,1)	B (1,2)	C (1,3)
D (2,1)	E (2,2)	F (2,3)
G (3,1)	H (3,2)	I (3,3)

FIG. 6.1 : Découpage d'un array à 3×3 macro blocs

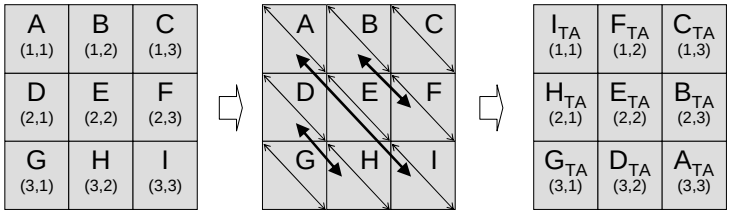
Une fois l'array découpé, les opérations de transposition ou de rotation vont se diviser en deux sous-problèmes :

- transposition/rotation à l'échelle des macro blocs et,
- transposition/rotation à l'intérieur de chacun des macro blocs.

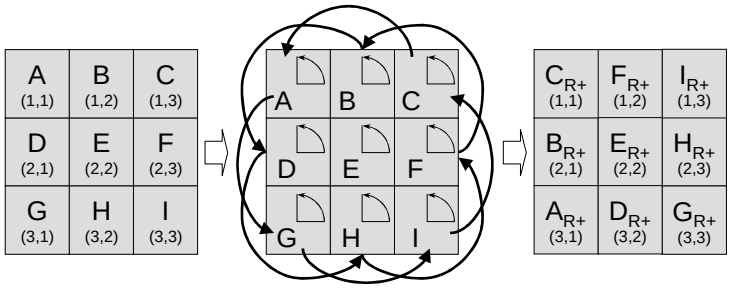
En effet, les transpositions/rotations partielles à l'intérieur des macro blocs sont des tâches indépendantes et peuvent être exécutées ainsi. La transposition à l'échelle globale qui manipulera les macro blocs va assurer la même opération au niveau de la granularité crue. La figure 6.2 illustre cette philosophie.



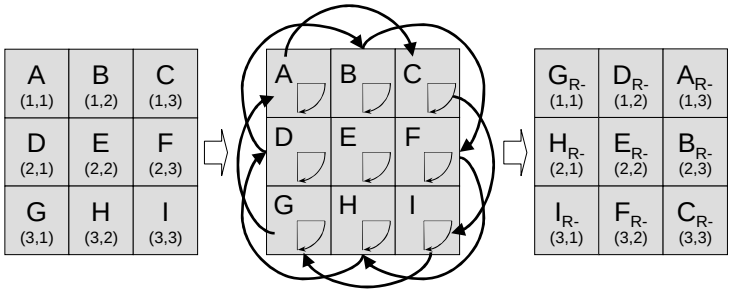
(a) Transposition d'un array par la diagonale



(b) Transposition d'un array par l'antidiagonale



(c) Rotation d'un array de $+\frac{\pi}{2}$



(d) Rotation d'un array de $-\frac{\pi}{2}$

FIG. 6.2 : Transpositions et rotations d'un array utilisant l'approche macro blocs

L'approche de travail par les macro blocs est générale et peut être implémentée sur n'importe quelle architecture, parallèle ou séquentielle. Nous nous en servons sur les architectures SWAR et nous allons en découvrir l'utilité pratique en exploitant les capacités SIMD, ce qui aboutira aux algorithmes plus rapides. Des performances encore meilleures peuvent être obtenues sur les architectures où nous pouvons exploiter le parallélisme de données et de tâche, implicitement présent dans ces algorithmes, e.g. sur les architectures multithread.

6.2.1 Découpage des arrays en macro blocs et leur recollage

Commençons notre explication du fonctionnement par macro blocs avec l'introduction formelle du découpage et du recollage des arrays.

Nous avons besoin de définir une fonction de découpage d'un array 2D en arrays 2D plus petits, tous ayant des dimensions identiques. La fonction arrayToMxNBlocs définit ce découpage. Elle prend comme arguments un array et deux paramètres, m et n, qui indiquent le nombre de macro blocs voulus dans l'array de sortie, respectivement dans la première et deuxième dimension. La fonction retourne un array de dimensions (m, n) dont les éléments sont également des arrays :

```
arrayToMxNBlocs          ::    I → I → Ar (I,I) α → Ar (I,I) (Ar (I,I) α)
arrayToMxNBlocs m n ar =
    array ((1,1), (m,n))
          [((i,j), array ((1,1), (p,q)) [((k,l), ar!( (i−1)*p+k,  (j−1)*q+l ))
                                              | (k,l) ← range ((1,1), (p,q))])
              |     (i,j) ← range ((1,1),(m,n))]
  where
    (d₁,d₂) = dimsAr2D ar;  p = div d₁ m; q = div d₂ n
```

Nous définissons également une fonction de composition, duale à la fonction de découpage, qui construit un array complet à partir d'un array découpé en macro blocs. La fonction arrayFromMxNBlocs effectue cette tâche :

```
arrayFromMxNBlocs    ::    Ar (I,I) (Ar (I,I) α) → Ar (I,I) α
arrayFromMxNBlocs ar =
    array ((1,1), (p,q))
          [(((i−1)*d₁+k, (j−1)*d₂+l), ar!(i,j)!(k,l))
              |     (i,j) ← range ((1,1), (m,n)), (k,l) ← range((1,1), (d₁,d₂)) ]
  where
    (m,n) = dimsAr2D ar; (d₁,d₂) = dimsAr2D (ar!(1,1)); p = d₁*m; q = d₂*n
```

6.2.2 L'algorithme générique travaillant sur les macro blocs

Nous avons déjà mentionné, d'une façon informelle, comment les algorithmes de transposition et de rotation vont travailler sur les macro blocs. Ce qui nous reste à détailler maintenant, c'est la représentation formelle d'un algorithme qui décrirait ce procédé.

Nous nous apercevons que les algorithmes de transposition (par la diagonale et antidiagonale) et de rotation (de $+\frac{\pi}{2}$ et de $-\frac{\pi}{2}$) sont très similaires dans leur fonctionnement. En effet, la structure de ces algorithmes est la même, ce qui change c'est l'opération exacte qui est effectuée. Nous allons en profiter pour introduire un algorithme qui assure la structure de fonctionnement de ces quatre algorithmes et qui est décrit par l'algorithme 6.2 comme fonction trRot2DMB.

Pour définir une opération spécifique, nous devons choisir les valeurs des arguments de ce skeleton. Les paramètres m et n désignent les nombres de blocs dans le premier et le deuxième axe sur lesquelles l'array d'entrée ar sera découpé. Le paramètre how est un paramètre textuel et désigne l'opération à exécuter. Les valeurs que nous pouvons passer par ce paramètre sont $"TD"$, $"TA"$, $"R+"$ et $"R-"$ et correspondent aux clefs de la fonction de définition commune pour les transpositions et les rotations, trRot2D. C'est en utilisant cette fonction que nous définissons les fonctions globales, fg, et locales, fl, à l'intérieur de cet algorithme.

Algorithme 6.2 : trRot2DMB, algorithme de la transposition/rotation par macro blocs. Les paramètres *m* et *n* désignent en combien de macro blocs l'array d'entrée *ar* sera découpé pour le travail, le paramètre *how* définit l'opération souhaitée.

```
1  trRot2DMB                    :: [Char] → I → I → Ar (I,I) α → Ar (I,I) α
2  trRot2DMB  how  m  n  ar =
3             arrayFromMxNBlocs
4           ∘ fg
5           ∘ (listArray ((1,1),(m,n)))
6           ∘ (map fl)
7           ∘ elems
8           ∘ (arrayToMxNBlocsmn)
9           $ ar
10    where
11       fg = trRot2D how
12       fl = trRot2D how
```

Expliquons son fonctionnement. Il est, en effet, assez simple. Nous commençons la lecture sur la ligne 9 où nous voyons l'array d'entrée *ar* et nous allons continuer vers les lignes précédentes. Nous appliquons sur cet array la fonction de découpage arrayToMxNBlocs (sur la ligne 8). Ainsi, nous obtenons un array 2D dont les éléments sont des arrays 2D plus petits qui constituent nos macro blocs. Après l'application de la fonction **elems**, standard du Haskell, nous obtenons un stream des macro blocs, exprimé comme une liste. Sur la ligne 6, nous appliquons la fonction locale *fl* sur tous les éléments de ce stream en utilisant la fonction **map** du Haskell. Sur la ligne 5, en utilisant la fonction **listArray**, nous reconstituons un array de $M \times N$ macro blocs à partir du stream des éléments pour y appliquer, sur la ligne 4, la fonction globale *fg*. À la fin, sur la ligne 3, nous passons, après l'application de la fonction arrayFromMxNBlocs, à partir d'un array des macro blocs à un array classique 2D qui est retourné par le skeleton comme résultat.

Notons que selon la prescription de cet algorithme, nous appliquons d'abord la fonction locale *fl* et puis la fonction globale *fg*. Mais l'approche duale peut être également effectuée avec d'abord l'application de la fonction globale *fg* et puis la fonction du grain fin *fl*, exprimé formellement par le changement des lignes 4 à 7 dans l'algorithme 6.2 pour :

```
4           ∘ (listArray $ ((1,1),(m,n)))
5           ∘ (map fl)
6           ∘ elems
7           ∘ fg
```

Cette manipulation est plutôt théorique et philosophique pour les architectures GPP car pour elles, c'est notre manière d'indexer lors des lectures/écritures des données de/à la mémoire qui assure la fonction globale. En revanche, le changement de ces lignes peut correspondre, sur les architectures matérielles dédiées, à la modification du réseau d'interconnexions. Remarquons que les deux possibilités délivrent les mêmes résultats.

Utilisant le skeleton algorithmique décrit précédemment, les définitions des transpositions et des rotations des arrays par macro blocs sont faciles à dériver. La fonction tr2DDiagMB définit la transposition d'un array par la diagonale en utilisant l'approche macro blocs et travaille à l'intérieur des macro blocs élément par élément :

```
tr2DDiagMB           :: I → I → Ar (I,I) α → Ar (I,I) α
tr2DDiagMB  m  n  ar = trRot2DMB "TD"  m  n  ar
```

La fonction tr2DADiagMB définit la transposition d'un array par l'antidiagonale en utilisant l'approche macro blocs et travaille à l'intérieur des macro blocs élément par élément :

```
tr2DADiagMB          :: I → I → Ar (I,I) α → Ar (I,I) α
tr2DADiagMB  m  n  ar = trRot2DMB "TA"  m  n  ar
```

La fonction rot2DPlus90MB définit la rotation d'un array de $+\frac{\pi}{2}$ en utilisant l'approche macro blocs et travaille à l'intérieur des macro blocs élément par élément :

```
rot2DPlus90MB           :: I → I → Ar (I,I) α → Ar (I,I) α
rot2DPlus90MB m n ar = trRot2DMB "R+" m n ar
```

La fonction rot2DMinus90MB définit la rotation d'un array de $-\frac{\pi}{2}$ en utilisant l'approche macro blocs et travaille à l'intérieur des macro blocs élément par élément :

```
rot2DMinus90MB           :: I → I → Ar (I,I) α → Ar (I,I) α
rot2DMinus90MB m n ar = trRot2DMB "R−" m n ar
```

6.3 Algorithmes rapides SIMD de transposition et de rotation

Nous allons nous intéresser aux algorithmes qui exploiteraient les possibilités des architectures SIMD et qui rendraient les transpositions et les rotations plus rapides qu'une approche directe et triviale de la transposition ou rotation effectuée élément par élément.

Mais avant de décrire les algorithmes SIMD destinés aux architectures GPPMM, nous allons présenter un outil qui va nous servir dans tout ce chapitre - les fonctions shuffle.

6.3.1 Fonctions shuffle

Les *shuffles* sont les fonctions de réarrangement des éléments dans les vecteurs. Dans notre cas, elles font alterner les éléments de deux vecteurs paquetés selon un paramètre définissant la façon exacte de leur chevauchement et les placent dans un nouveau vecteur paqueté de sortie. Ainsi, on les catégorise comme les cas spéciaux de la permutation des éléments d'un vecteur. Puisque notre intérêt est de rester abstrait d'une architecture quelconque, nous introduisons les fonctions généralisées du *low-shuffling* et du *hi-shuffling* qui, même généralisées, ont leurs correspondants directs dans tous les jeux d'instructions des architectures multimédia.

Dans les définitions des *shuffles*, nous utilisons deux fonctions auxiliaires, ielems et alter. La fonction ielems prend un vecteur et une étendue d'index et retourne les éléments inclus dans cette étendue :

```
ielems          :: (Ix α) ⇒ Ar α β → (α,α) → [β]
ielems ar rng = [ar!i | i ← range$rng]
```

La fonction alter crée, à partir de deux listes xs et ys de la même taille N, une seule liste de sortie de la taille $2N$ en alternant les entrées. Le paramètre n règle l'alternance de cette manière : les premiers n éléments de la première liste sont inscrits comme premiers dans la liste de sortie et suivis par n premiers éléments de la deuxième liste. Ce processus se répète pour tous les groupes suivants de n éléments des listes d'entrée. Notons que la taille N des listes d'entrée doit être un multiple de n.

```
alter :: I → ([α],[α]) → [α]
alter n (xs,ys) = select n [] (xs,ys)
    where
        select k zs (x:xs,ys) | k > 0  = select (k−1) (zs++[x]) (xs,ys)
        select k zs (xs,ys)   | k == 0 = select n zs (ys,xs)
        select k zs ([],_)             = zs
```

Le *low-shuffling* est défini par la fonction shflo. Les moitiés inférieures des deux vecteurs d'entrée sont extraites en utilisant la fonction ielems. Leurs éléments sont alternés, en utilisant la fonction alter, avec l'alternance n commençant avec le premier vecteur. La fonction dimsAr1D, définie en Annexe B, page 202, retourne la taille d'un vecteur.

```
shflo :: I → (PVec I α, PVec I α) → PVec I α
shflo s (x,y) = listArray (1,p) (alter s ((ielems x (1, h)),
                                           (ielems y (1, h))))
    where p = dimsAr1D x;  h = div p 2
```

Par dualité, le *hi-shuffling*, est défini par la fonction shfhi. Il assure la fonctionnalité similaire sur les moitiés supérieures des deux vecteurs d'entrée.

```
shfhi  ::  I → (PVec I α, PVec I α) → PVec I α
shfhi s (x,y) = listArray  (1,p)  (alter s ((ielems  x (h+1, p)),
                                            (ielems  y (h+1, p)) ))
   where p = dimsAr1D x;    h = div p 2
```

Notons que les arrays d'entrée des fonctions shuffle shflo et shfhi doivent avoir la même dimension et celle-ci doit être de $2^m, m \in \mathbb{N}$. La figure 6.3 illustre la gamme complète des shuffles dérivables à partir des définitions précédentes pour les arrays de 2^3 éléments.

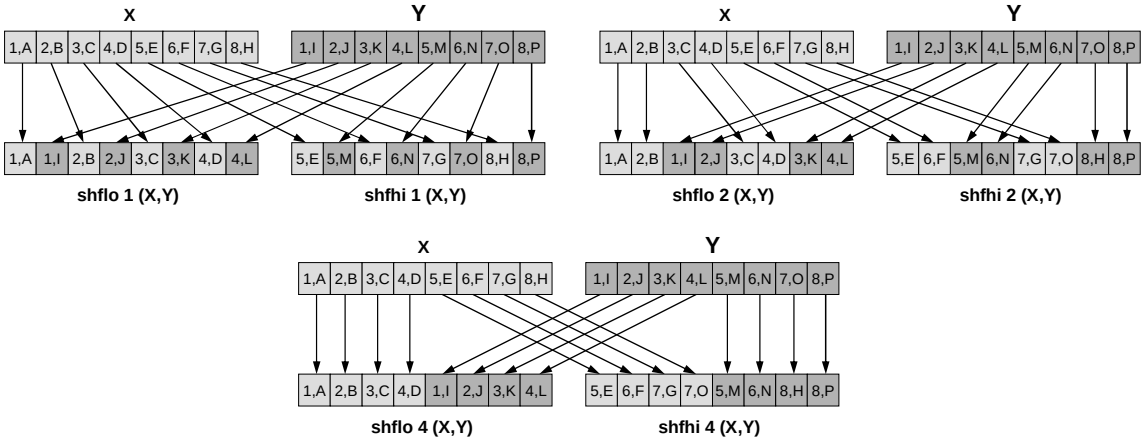


FIG. 6.3 : La gamme des fonctions shuffle pour les vecteurs paquetés de 8 éléments

6.3.2 Découpage sur les macro blocs et leur recollage sur les architectures SWAR

Dans notre travail SIMD avec les architectures SWAR, il est convenable de découper l'array d'entrée en macro blocs dont les dimensions sont égales à la taille N du registre qui héberge les données des types multimédia. Les processeurs GPPMM les plus courants (SH-5, Intel IA-64, AMD64, Intel MMX) ont la taille du registre multimédia de 64 bits ce qui prédestine, pour le travail avec les données de 8 bits, le découpage en macro bloc de 8×8 éléments. Le type de 8 bits est couramment utilisé dans le traitement d'images comme le type de base pour le stockage des pixels. Si nous utilisons le type de 16 bits, notre macro bloc serait réduit à 4×4 éléments.

Sur les architectures différentes avec les registres plus larges, telles qu'Intel SSE2/3 dont les registres sont de 128 bits, le choix du macro bloc peut être différent. Notons que ces architectures ont la taille de leurs registres d'une puissance n de 2, $N = 2^n$, ce qui satisfait les exigences des fonctions shuffles sur les dimensions des vecteurs d'entrée.

En plus du choix de découpage, nous allons accéder aux données dans la mémoire en utilisant les types paquetés. Ainsi, nous allons percevoir les arrays dans nos définitions comme paquetés. Ce qui veut dire qu'à la place de travailler avec les macro blocs de dimensions $N \times N$ du type

```
Ar (I,I) α
```

nous allons travailler, après le paquetage, comme défini dans 4.4.3 page 68, avec les macro blocs du type

```
Ar (I,I) (PVec I α)
```

qui auront les éléments du type PVec de taille N et où une dimension de cet array sera réduite à 1. Ainsi, nous allons travailler avec un array d'une dimension qui sera stocké dans un type pour les arrays 2D. Celle des dimensions qui sera réduite à 1 dépendra de notre choix du sens du paquetage. Cette perception est, en effet, un cas spécial issu du choix particulier des dimensions du macro bloc.

Par la suite, nous allons travailler avec ces macro blocs comme s'il s'agissait de flux de données. Pour passer d'un macro bloc à un stream et vice versa, nous pourrions utiliser les fonctions génériques que nous avons définies dans ce but dans le chapitre 4.4.4, page 70. Mais vu que le sens de parcours dont nous avons besoin ici est simple et se réduit au sens *au-devant*, nous allons utiliser deux fonctions standard du Haskell, **elems** et **listArray**. La première, **elems** retourne une liste de tous les éléments d'un array, la deuxième, **listArray** construit un array à partir d'une liste des éléments. Nous allons les utiliser dans des expressions pour passer d'un array 2D exprimé par la variable *ar* à un stream comme :

elems$*ar*

et pour passer à partir d'un stream *ss* à un array dont les dimensions seront les mêmes que celles du array *ar* comme :

listArray (**bounds** $ *ar*) *ss*

C'est la même manière de travailler que nous avons déjà mentionnée dans l'algorithme général par macro blocs, cf. l'algorithme 6.2, et elle est suffisante et assez intuitive pour les explications qui sont décrites dans la suite de ce chapitre.

6.3.3 Shuffles utilisés pour les transpositions et rotations d'un macro bloc

Notre approche *par macro blocs* dont la taille est dérivée de la taille du registre multimédia d'une architecture nous mène à une application très intéressante des fonctions shuffle. En choisissant leur bonne combinaison et leur bon enchaînement suivi par l'application sur les données paquetées d'un macro-bloc, nous pouvons assurer les quatre opérations qui nous intéressent (transposition par diagonale/antidiagonale, rotation de $+\frac{\pi}{2}$ et de $-\frac{\pi}{2}$).

Pour présenter une explication claire et facile à comprendre, nous allons expliquer l'utilisation des shuffles sur un exemple précis de la transposition par diagonale d'un macro bloc 8×8. Une fois le procédé expliqué, nous continuerons par la généralisation de cette approche aux macro blocs de dimensions $N \times N$, où $N = 2^n$, qui inclura également les trois opérations restantes.

6.3.3.1 Transposition par diagonale avec les shuffles

Expliquons en détail le fonctionnement de l'algorithme de la transposition par diagonale d'un macro bloc de 8×8 éléments. L'algorithme travaille sur un macro bloc exprimé en tant que liste (stream) de 8 vecteurs paquetés qui comptent 8 éléments chacun. Il applique sur ce stream des groupes de différentes fonctions shuffle. Ainsi, nous percevons le traitement comme divisé en étapes, en total $\log_2 N$ étapes, ce qui implique 3 étapes pour le bloc 8×8. La figure 6.4 illustre cette situation

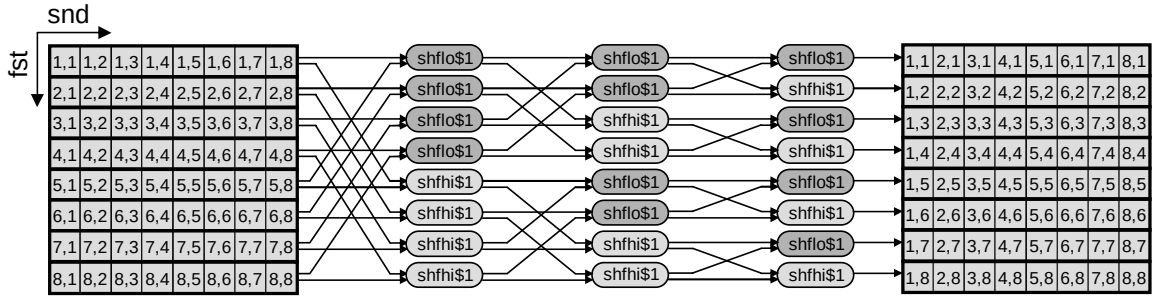


FIG. 6.4 : Transposition par diagonale SIMD par shuffles

L'algorithme 6.3 décrit formellement ce procédé. Il définit une fonction principale, tr2DDiag8x8bf, dans le corps de laquelle nous retrouvons 3 applications successives des blocs composés de plusieurs shuffles, présentées par les application successives de la fonction tr2DDiag8x8bf1, puis de la fonction tr2DDiag8x8bf2 et finalement de la fonction tr2DDiag8x8bf3. Chaque étape utilise plusieurs shuffles

Algorithme 6.3 : tr2DDiag8x8bf, Algorithme de tranposition d'un macro bloc 8×8 qui utilise 3 shuffles partiels (tr2DDiag8x8bf1, tr2DDiag8x8bf2, tr2DDiag8x8bf3)

```
1  tr2DDiag8x8bf     ::  [PVec I α] → [PVec I α]
2  tr2DDiag8x8bf ss = tr2DDiag8x8bf3 ∘ tr2DDiag8x8bf2 ∘ tr2DDiag8x8bf1 $ ss
3
4  tr2DDiag8x8bf1    ::  [PVec I α] → [PVec I α]
5  tr2DDiag8x8bf1 xs = alter 4 ((map (shflo $1) pairs), (map (shfhi $1) pairs))
6      where  pairs = listDivAlter 4 xs
7
8  tr2DDiag8x8bf2    ::  [PVec I α] → [PVec I α]
9  tr2DDiag8x8bf2 xs = alter 2 ((map (shflo $1) pairs), (map (shfhi $1) pairs))
10     where  pairs = listDivAlter 2 xs
11
12 tr2DDiag8x8bf3    ::  [PVec I α] → [PVec I α]
13 tr2DDiag8x8bf3 xs = alter 1 ((map (shflo $1) pairs), (map (shfhi $1) pairs))
14     where  pairs = listDivAlter 1 xs
```

dans une configuration différente. À la fin du traitement, nous obtenons le macro bloc, exprimé par la liste des vecteurs paquetés, transposé par la diagonale.

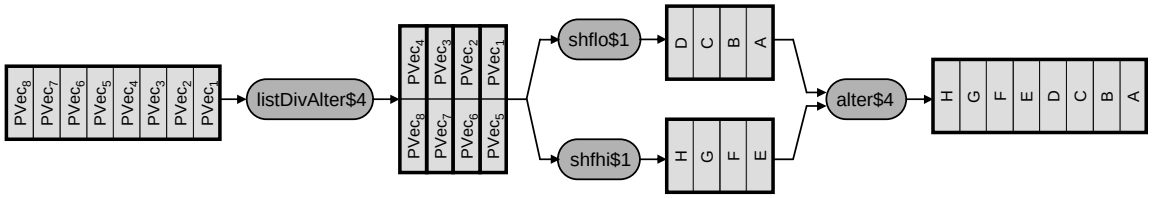


FIG. 6.5 : Illustration du fonctionnement de la fonction tr2DDiag8x8bf1

Dans les 3 fonctions partielles, la variables *pairs* contient la liste dont les éléments sont les arguments pour les fonctions shuffles. C'est notre fonction utilitaire listDivAlter qui se charge de bon ordonnancement pour obtenir une liste des paires (PVec I α, PVec I α) qui sont prêtes à être employées sur les fonctions shuffles. La fonction alter nous sert ici à la bonne organisation des résultats dans le flux de sortie. La fig. 6.5 illustre ce processus pour la fonction tr2DDiag8x8bf1.

En effet, cette façon de diviser des streams d'entrée et leur alternance exprime formellement le réseau d'interconnexions particulier entre les données et les blocs exécutifs. Ce réseau est également connu comme des *papillons* (angl. *butterflies*) et il est illustré sur la fig. 6.4. Notons que les flèches supérieures resp. inférieures à l'entrée de chacun des blocs des opérations shuffles désignent les premiers resp. deuxièmes éléments dans les tuples qui constituent les arguments d'entrée de ces opérations.

La fonction alter était déjà présentée, page 133, mais la définition exacte de la fonction listDivAlter est nouvelle :

```
listDivAlter              ::  I → [α] → [(α,α)]
listDivAlter     n xs     =  select n ([],[]) xs
  where
    select k (lo,hi) (x:xs)   | k > 0   =  select (k−1) (lo++[x],hi) xs
    select k (lo,hi) xs       | k == 0  =  select n (hi,lo) xs
    select k (lo,hi) []                  =  (uncurry$zip) (lo, hi)
```

Cette fonction définit le réseau d'interconnexions. Pour une liste d'entrée xs dont la taille est $N = 2^k, k \geq 1$, cette fonction crée une liste de taille $\frac{N}{2}$ dont les éléments sont des tuples. La création de cette liste de sortie passe par une phase intermédiaire où la liste d'entrée xs est divisée d'une façon alternée en deux listes, chacune de taille $\frac{N}{2}$, qui ne sont pas explicitement mentionnées dans la définition mais

que nous pouvons nommer ys_1 et ys_2 et qui sont placées dans un tuple (ys_1, ys_2) et sur lesquelles nous exécutons récursivement la fonction interne select. Ainsi, les éléments de la liste xs sont distribués en alternance vers les deux listes ys_1 et ys_2, les premiers n éléments vers la liste ys_1, les n éléments suivants dans la liste ys_2, les n suivants vers la liste ys_1, etc... Quand nous arrivons à la fin de la liste d'entrée, la liste des tuples de sortie est créée en utilisant l'expression (**uncurry**\$**zip**).

Pour mieux comprendre l'emploi de cette fonction, nous concrétisons la définition du stream des tuples des vecteurs paquetés *pairs* dans l'algorithme 6.3. Pour un stream concret $[1, 2, 3, 4, 5, 6, 7, 8]$ nous obtenons :
après l'application de *listDivAlter*\$4 sur ce stream (comme dans tr2DDiag8x8bf1) :

pairs = [(1,5), (2,6), (3,7), (4,8)]

après l'application de *listDivAlter*\$2 sur ce stream (comme dans tr2DDiag8x8bf2) :

pairs = [(1,3), (2,4), (5,7), (6,8)]

après l'application de *listDivAlter*\$1 sur ce stream (comme dans tr2DDiag8x8bf3) :

pairs = [(1,2), (3,4), (5,6), (7,8)]

Ce qui correspond au réseau d'interconnexions des papillons pour cette opération, comme présenté sur la fig. 6.4.

Les définitions des fonctions tr2DDiag8x8bf1, tr2DDiag8x8bf2 et tr2DDiag8x8bf3, bien qu'elles soient explicites, utilisent une prescription qui énumère les éléments. Nous pouvons apercevoir également qu'elles sont très semblables car le cœur de leur fonctionnement est donné par une et même expression :

```
((map (shflo $1) pairs), (map (shfhi $1) pairs))
```

Ils diffèrent dans la manière de composer le stream des paires d'entrée *pairs* par la fonction listDivAlter, mais également dans la composition du stream de sortie par la fonction alter.

Nous allons exploiter des points communs et nous allons récrire ces définitions d'une façon encore plus simple et généralisée non seulement pour les macro blocs de 8×8 éléments, mais également pour un cas général d'un macro bloc de $N \times N$ éléments, $N = 2^n$ et $n = 1, 2, 3, ...$ Ainsi, nous arrivons à la définition de l'algorithme généralisé qui n'utilise que deux fonctions, tr2DDiagNxPVecNbfi et de tr2DDiagNxPVecNbf et qui est défini par l'algorithme 6.4 :

Algorithme 6.4 : tr2DDiagNxPVecNbf, fonction généralisée de transposition d'un macro bloc par diagonale. Elle utilise en interne la fonction tr2DDiagNxPVecNbfi

```
1  tr2DDiagNxPVecNbf      ::  [PVec I α] → [PVec I α]
2  tr2DDiagNxPVecNbf ss =  pipe ( map tr2DDiagNxPVecNbfi rs )  $ ss
3      where  n = rangeSize ∘ bounds  $  (ss!!0);   rs = seqPow2 (div n 2)
4
5  tr2DDiagNxPVecNbfi          ::  I → [(PVec I α)] → [(PVec I α)]
6  tr2DDiagNxPVecNbfi p xs
7          = alter p ((map (shflo $1) pairs), (map (shfhi $1) pairs))
8      where   pairs = listDivAlter p xs
```

Une seule fonction, tr2DDiagNxPVecNbfi, définit maintenant tous les groupes des shuffles pour la transposition par diagonale. Cette fonction modifie son comportement selon le paramètre p choisissant ainsi la bonne configuration des éléments d'entrée. La valeur de p est une des valeurs de séquence rs des puissances de 2 définie par la fonction seqPow2.

```
seqPow2  ::  Int → [Int]
seqPow2 x =  fnc  x []
    where
        fnc  k xs  |  k ≥ 1   =  fnc  (div k 2) ([k]++xs)
        fnc  k xs  |  k == 0  =  xs
```

Cette fonction retourne une liste des nombres de puissance 2. La liste contient des numéros commençant par 1 et allant jusqu'à l'égalité avec le paramètre *x*, e.g. $[1, 2, 4, 8, 16, 32]$ pour la valeur de *x* égale à 32. La valeur du paramètre doit être une puissance de 2.

La fonction tr2DDiagNxPVecNbf définit tout l'algorithme et applique $\log_2 N$ étapes de shuffles bien choisis en utilisant la fonction pipe. Tout le travail est caché dans l'expression

```
pipe (map tr2DDiagNxPVecNbfi rs)
```

Une liste *rs* est créée par la fonction seqPow2 et contient les valeurs $2^0, 2^1, ..2^{m-1}, 2^m$ où m est une puissance de 2 qui est, dans ce cas, égale à $m = \log_2 N - 1$. Ainsi, l'expression peut être récrite, d'une façon informelle et après l'application de map, comme une liste :

```
pipe ([tr2DDiagNxPVecNbfi$1,
       tr2DDiagNxPVecNbfi$2,
       .. ,
       tr2DDiagNxPVecNbfi$(log₂N−1)])
```

Ainsi, nous avons décrit d'une façon générale, le fonctionnement de la transposition par diagonale pour un macro bloc de NxN éléments, $N = 2^n$ et $n = 1, 2, 3, ..$, organisé comme une liste dont les éléments sont les vecteurs paquetés.

De la même façon, nous allons définir les algorithmes pour la transposition par l'antidiagonale, pour la rotation de $\frac{\pi}{2}$ et de $-\frac{\pi}{2}$. L'approche est directe et mène aux définitions dans lesquelles nous allons remarquer le changement mutuel, soit des fonctions shuffle, soit de l'ordre dans la préparation des arguments pour les shuffles, soit des deux.

6.3.3.2 Transposition par antidiagonale avec les shuffles

La transposition par antidiagonale d'un macro bloc qui utilise les shuffles est définie par la fonction tr2DADiagNxPVecNbf qui utilise à l'interne la fonction tr2DADiagNxPVecNbfi. Nous remarquons que cette définition a la même structure que la transposition par diagonale, définie par la fonction tr2DDiagNxPVecNbf dans l'algorithme 6.4.

```
1  tr2DADiagNxPVecNbf    ::  [PVec I α] → [PVec I α]
2  tr2DADiagNxPVecNbf ss =   pipe ( map tr2DADiagNxPVecNbfi rs) $ ss
3      where  n = rangeSize ∘ bounds $ (ss!!0);   rs = seqPow2 (div n 2)
4
5  tr2DADiagNxPVecNbfi       ::  I → [(PVec I α)] → [(PVec I α)]
6  tr2DADiagNxPVecNbfi p xs
7          = alter p ((map (shfhi $1) (xchng$pairs)), (map (shflo $1) (xchng$pairs)))
8      where   pairs = listDivAlter p xs
```

Ce qui a changé, c'est l'expression sur la ligne 7, où nous pouvons percevoir l'échange mutuel des fonctions shuffle shfhi$1 et shflo$1 et également l'application de la fonction xchng sur la variable *pairs* qui change l'ordre dans les arguments pour les fonctions shuffle. La définition exacte de la fonction xchng qui, pour une liste des tuples donnée, échange l'ordre dans tous les tuples de cette liste, est la suivante :

```
xchng             ::  [(α,β)] → [(β,α)]
xchng [ ]         =   [ ]
xchng ((x,y):ss)  =   (y,x):(xchng ss)
```

6.3.3.3 Rotation de $+\frac{\pi}{2}$ avec les shuffles

La rotation de $\frac{\pi}{2}$ peut être définie de la même manière. La fonction rot2DPlus90NxPVecNbf qui utilise à l'interne la fonction rot2DPlus90NxPVecNbfi a la même structure que les deux opérations précédentes. Un changement est présent, c'est celui de la ligne 7 qui échange mutuellement l'ordre des arguments des fonctions shuffle. Il est exprimé via l'application de la fonction xchng appliquée sur la variable *pairs*.

```
1  rot2DPlus90NxPVecNbf      ::  [PVec I α] → [PVec I α]
2  rot2DPlus90NxPVecNbf  ss =  pipe ( map rot2DPlus90NxPVecNbfi rs) $ ss
3      where  n = rangeSize ∘ bounds $ (ss!!0);   rs = seqPow2 (div n 2)
4
5  rot2DPlus90NxPVecNbfi       ::  I → [(PVec I α)] → [(PVec I α)]
6  rot2DPlus90NxPVecNbfi  p xs
7          = alter p ((map (shflo $1) (xchng$pairs)), (map (shfhi $1) (xchng$pairs)))
8      where  pairs = listDivAlter  p xs
```

6.3.3.4 Rotation de $-\frac{\pi}{2}$ avec les shuffles

De la même manière, nous définissons la rotation de $\frac{\pi}{2}$ par la fonction rot2DMinus90NxPVecNbf qui utilise à l'interne la fonction rot2DMinus90NxPVecNbfi. Ces définitions différent de la fonction de transposition par diagonale dans l'échange mutuel des fonctions shuffle shfhi$1 et shflo$1 sur la ligne 7.

```
1  rot2DMinus90NxPVecNbf     ::  [PVec I α] → [PVec I α]
2  rot2DMinus90NxPVecNbf ss =  pipe ( map rot2DMinus90NxPVecNbfi  rs) $ ss
3      where  n = rangeSize ∘ bounds $ (ss!!0);   rs = seqPow2 (div n 2)
4
5  rot2DMinus90NxPVecNbfi      ::  I → [(PVec I α)] → [(PVec I α)]
6  rot2DMinus90NxPVecNbfip xs
7          = alter p ((map (shfhi $1) pairs), (map (shflo $1) pairs))
8      where  pairs = listDivAlter  p xs
```

6.3.3.5 Algorithme généralisé de transposition et de rotation d'un macro bloc par les shuffles

Les similitudes que l'on a pu apercevoir dans les définitions précédentes des transpositions et des rotations par macro blocs nous incitent penser que l'on pourrait regrouper les quatre fonctions en un seul algorithme généralisé qui réutiliserait les parties qui se répètent.

En effet, c'est possible et l'algorithme 6.5 décrit cette généralisation. Comme nous pouvons le voir, la structure donnée par la fonction trRot2DNxPVecNbf reste la même, le fonctionnement de l'algorithme est modifié par le premier paramètre qui exprime le type d'opération à exécuter. Celui-ci est passé à la fonction interne de cet algorithme, trRot2DNxPVecNbfi, qui se charge du bon choix de l'opération. Les valeurs de ce paramètre, les abréviations "TD", "TA", "R+" et "R-", désignent respectivement la *transposition par diagonale*, la *transposition par antidiagonale*, la *rotation de* $\frac{\pi}{2}$ et finalement la *rotation de* $-\frac{\pi}{2}$.

Algorithme 6.5 : trRot2DNxPVecNbf, algorithme généralisé de transposition et rotation d'un array 2D *NxPVecN*, utilise la fonction trRot2DNxPVecNbfi

```
1  trRot2DNxPVecNbf            ::  [Char] → [PVec I α] → [PVec I α]
2  trRot2DNxPVecNbf  how  ss =  pipe ( map (trRot2DNxPVecNbfi$how) rs) $ ss
3      where  n = rangeSize ∘ bounds $ (ss!!0);   rs = seqPow2 (div n 2)
4
5  trRot2DNxPVecNbfi           ::  [Char] → I → [(PVec I α)] → [(PVec I α)]
6  trRot2DNxPVecNbfi  how  p  xs
7      | how == "TD"
8          = alter p ((map (shflo $1)        pairs), (map (shfhi $1)        pairs))
9      | how == "TA"
10         = alter p ((map (shfhi $1) (xchng$pairs)), (map(shflo $1)(xchng$pairs)))
11     | how == "R+"
12         = alter p ((map (shflo $1) (xchng$pairs)), (map(shfhi $1)(xchng$pairs)))
13     | how == "R−"
14         = alter p ((map (shfhi $1)        pairs), (map(shflo $1)        pairs))
15   where  pairs = listDivAlter  p xs
```

Ce skeleton algorithmique est important car il nous permet, par la suite, de définir facilement l'algorithme complet pour les quatre opérations en les distinguant par un seul paramètre.

6.3.4 Algorithme complet pour les transpositions et les rotations par SIMD

Une fois expliqué ce qui se passe à l'intérieur d'un macro bloc, nous allons détailler le processus pour les transpositions et les rotations des arrays. Pour cela, nous allons revenir à l'algorithme 6.2, présenté précédemment sur la page 132, qui décrivait le skeleton algorithmique pour les transpositions/rotations par macro blocs en travaillant élément par élément. Il va nous servir comme modèle pour la construction d'un nouvel algorithme.

Ce nouvel algorithme, que nous présentons ici comme l'algorithme 6.6, va utiliser pour son travail l'approche SIMD. Ainsi, l'accès aux données sera effectué en utilisant les types des vecteurs paquetés. Ce qui signifie que cet algorithme va percevoir l'array d'entrée comme un array dont les éléments sont du type des vecteurs paquetés PVec. Par la suite, il découpera cet array en macro blocs et il effectuera l'opération choisie localement à l'intérieur de chacun des macro blocs en utilisant, bien sûr, les algorithmes SIMD décrits précédemment. Puis il effectuera la même opération globalement avec les macro blocs en utilisant l'algorithme de base issue de la définition de cette opération.

Algorithme 6.6 : trRot2DMBSIMD, algorithme complet de la transposition et rotation d'un array 2D utilisant l'approche macro bloc et les fonctionnalités SIMD

```
1  trRot2DMBSIMD :: [Char] → [Char] → I → Ar (I,I) α → Ar (I,I) α
2  trRot2DMBSIMD how axe mbsize ar =
3              (mkAr2DFromAr2DPVec axe)
4          ∘ arrayFromMxNBlocs
5          ∘ fg
6          ∘ (listArray ((1,1),(m,n)))
7          ∘ (map (listArray ((1,1),(1,mbsize)) ))
8          ∘ (map fl)
9          ∘ (map elems)
10         ∘ elems
11         ∘ (arrayToMxNBlocs m n )
12         ∘ (mkAr2DPVec axe mbsize)
13         $ ar
14     where
15         (p,q) = dimsAr2D ar; (m,n) = ((div p mbsize),(div q mbsize))
16         fg = trRot2D how
17         fl = trRot2DNxPVecNbf how
```

Les paramètres de cet algorithme sont les suivants : *how* désigne le type d'opération à effectuer et peut avoir les valeurs *"TD"* pour la transposition par diagonale, *"TA"* pour la transposition par l'antidiagonale, *"R+"* pour la rotation de $+\frac{\pi}{2}$ et *"R−"* pour la rotation de $-\frac{\pi}{2}$. Le paramètre *axe* désigne le sens de vectorisation et peut avoir les valeurs *"Fst"* pour le premier axe et *"Snd"* pour le deuxième. Le paramètre *mbsize* désigne la taille des macro blocs et *ar* désigne l'array d'entrée.

Expliquons alors, pas à pas, la construction exacte de cet algorithme. La lecture commence sur la ligne 13 et on va progresser vers les lignes précédentes. La première étape est constituée du passage d'un array *ar* (ligne 13) avec les éléments du type α à un array avec les éléments paquetés PVec I α. Pour effectuer cela, nous allons utiliser la fonction mkAr2DPVec (ligne 12) avec la bonne clé, soit *"Fst"*, soit *"Snd"*. Le choix du sens de la vectorisation est prédéfini par l'axe de stockage des données dans la mémoire. Ensuite, en utilisant la fonction **elems**, nous extrayons tous les éléments de cet array vectorisé et nous les plaçons dans un stream (ligne 10). Pour pouvoir appliquer les fonctions macro blocs SIMD comme décrites précédemment, nous devons passer, pour chacun des macro blocs, à son expression en

tant que stream. C'est effectué par le mapping (**map elems**) sur la ligne 9. Nous obtenons ainsi un stream des streams, formellement décrit comme :

$$[[\mathsf{PVec}\ \mathsf{I}\ \alpha]]$$

Sur la ligne 8, nous appliquons la fonction locale par l'expression (**map** *fl*) à chacun des macro blocs exprimés en stream. Ensuite, sur la ligne 7, nous passons à l'expression des macro blocs en tant que array *2D* et nous reconstituons à nouveau un array dont les éléments sont les macro blocs sur la ligne 6. L'opération globale, *fg*, est appliquée sur cet array reconstitué (sur la ligne 5) achevant ainsi notre opération. Ce qui reste à faire c'est de passer à partir d'un array des macro blocs à un array dont les éléments sont les vecteurs paquetés (sur la ligne 4) pour, à la fin, appliquer une opération inverse à la vectorisation qui donne comme résultat un array du même type que celui d'entrée de la fonction, du type $\mathsf{Ar}\ (\mathsf{I}, \mathsf{I})\ \alpha$.

Ainsi, nous avons obtenu l'opération souhaitée en utilisant l'approche macro bloc et en employant les opérations SIMD sur les macro blocs.

6.4 Notes sur l'implémentation, résultats expérimentaux

Il y a, en effet, autant de façons d'implémenter les algorithmes décrits dans ce chapitre qu'il y a d'architectures, de programmeurs pour l'écriture et de compilateurs pour la compilation du code.

Les implémentations sur les architectures parallèles peuvent être facilement déduites de nos descriptions formelles des algorithmes présentés dans ce chapitre. Le parallélisme le plus simple, utilisable dans ces cas, est celui de la *replication fonctionnelle* représentée par le skeleton algorithmique farm, cf. 4.4.2.1, page 67. Pour l'employer, nous nous intéressons à toutes les parties de notre algorithme qui utilisent la fonction **map** de l'application d'une fonction sur tous les éléments d'un stream. Toutes ces parties peuvent être récrites en utilisant le skeleton algorithmique farm à la place de la fonction **map**. Ainsi, nous changeons complètement la manière de travailler d'une telle partie de notre algorithme et nous passons de l'exécution en séquence, exprimée par **map**, à l'exécution en parallèle, exprimée par farm. Le choix exact dépend de nos exigences et de nos possibilités matérielles lors de l'implémentation.

De plus, ces algorithmes entrent dans la logique du paradigme *Divide and Conquer*, présenté par le skeleton algorithmique dc, cf. 4.4.2.2, page 67. La division d'un problème global à des problèmes plus petits et locaux est propre aux algorithmes de ce chapitre travaillant sur les macro blocs. Il serait également envisageable d'exprimer ces algorithmes en termes du *Divide and conquer* et en utilisant le skeleton algorithmique dc car la manière de travailler de ce skeleton est identique à ce que nous faisons par le découpage d'un array sur les macro blocs, l'application de la fonction locale et son recollage effectué à la fin.

Concernant l'implémentation SIMD, la première chose que nous devrions souligner est la demande d'alignement des données de l'image dans la mémoire aux bornes qui sont les multiples de la taille N du registre multimédia. Si l'image a des dimensions qui sont des multiples de N et si, de plus, elles est alignée aux blocs de mémoire par N, notre implémentation se révèle simple. Dans le cas contraire, nous devrions faire face aux effets particuliers du travail avec les données non-alignées. L'accès aux données non-alignées est possible sur les architectures multimédia via les instructions spécialisées pour un accès non-aligné mais le coût d'un tel accès est, en général, supérieur à un accès alignée. C'est du au fait que pour la lecture d'une donnée non-alignée vers un registre, l'architecture utilise deux lectures consécutives des zones alignées couvrant les données voulues suivies par leur extraction vers le registre. Ces instructions peuvent avoir un coût relativement faible, mesuré dans les cycles d'horloge, comme c'est le cas pour les instructions Intel SSE3. La figure 6.7 illustre un exemple de la transposition d'une image alignée mais dont les dimensions ne sont pas un multiple de la taille du registre multimédia.

Nous présentons également deux exemples du code en langage C implémentant la transposition d'un macro bloc par la diagonale.

Le premier, présenté sur la fig. 6.6, est un code qui provient du MorphoMedia, un outil logiciel que nous avons développé dans le cadre de cette thèse. Il s'agit d'un code programmé comme les directives du préprocesseur (cf. `#define`) qui utilise les fonctions commençant par `mrph_asm_` qui nous

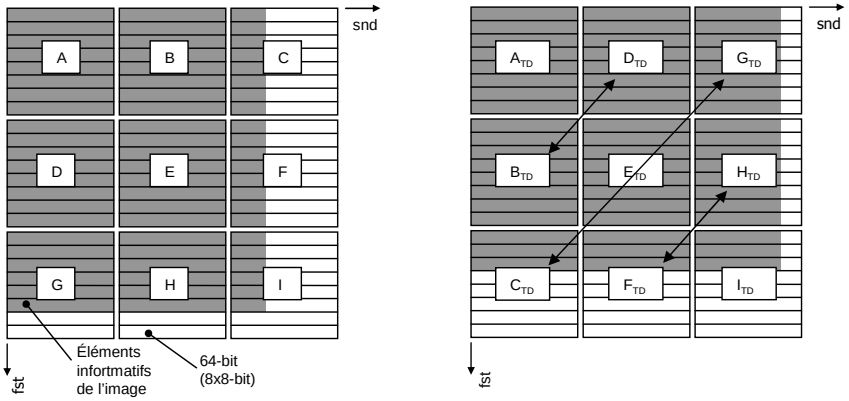


FIG. 6.7 : La transposition d'un array dont les dimensions ne sont pas un multiple de la taille d'un registre multimédia de 64 bits ; TD = macro bloc transposé par la diagonale

servent comme les invariables architecturales dans notre code. Ainsi, le même code peut être réutilisé sur plusieurs architectures multimédia.

```
#define MRPH_MACRO_TrByMainDiagonal_8x8_t8(\
    A0, A1, A2, A3, A4, A5, A6, A7, \
    B0, B1, B2, B3, B4, B5, B6, B7 \
  ) \
{\
      B0 = mrph_asm_mshflo_iu8vec8(A0, A4);\
      B1 = mrph_asm_mshflo_iu8vec8(A1, A5);\
      B2 = mrph_asm_mshflo_iu8vec8(A2, A6);\
      B3 = mrph_asm_mshflo_iu8vec8(A3, A7);\
      B4 = mrph_asm_mshfhi_iu8vec8(A0, A4);\
      B5 = mrph_asm_mshfhi_iu8vec8(A1, A5);\
      B6 = mrph_asm_mshfhi_iu8vec8(A2, A6);\
      B7 = mrph_asm_mshfhi_iu8vec8(A3, A7);\
 \
      A0 = mrph_asm_mshflo_iu8vec8(B0, B2);\
      A1 = mrph_asm_mshflo_iu8vec8(B1, B3);\
      A2 = mrph_asm_mshfhi_iu8vec8(B0, B2);\
      A3 = mrph_asm_mshfhi_iu8vec8(B1, B3);\
      A4 = mrph_asm_mshflo_iu8vec8(B4, B6);\
      A5 = mrph_asm_mshflo_iu8vec8(B5, B7);\
      A6 = mrph_asm_mshfhi_iu8vec8(B4, B6);\
      A7 = mrph_asm_mshfhi_iu8vec8(B5, B7);\
 \
      B0 = mrph_asm_mshflo_iu8vec8(A0, A1);\
      B1 = mrph_asm_mshfhi_iu8vec8(A0, A1);\
      B2 = mrph_asm_mshflo_iu8vec8(A2, A3);\
      B3 = mrph_asm_mshfhi_iu8vec8(A2, A3);\
      B4 = mrph_asm_mshflo_iu8vec8(A4, A5);\
      B5 = mrph_asm_mshfhi_iu8vec8(A4, A5);\
      B6 = mrph_asm_mshflo_iu8vec8(A6, A7);\
      B7 = mrph_asm_mshfhi_iu8vec8(A6, A7);\
}
```

FIG. 6.6 : Code de la transposition par diagonale d'un macro bloc 8×8 en langage C utilisant l'outil de développement multiplateforme MorphoMedia

Le deuxième exemple est présenté sur la fig. 6.8. Il s'agit d'un code écrit manuellement qui assure la même fonctionnalité de transposition d'un macro bloc par la diagonale mais qui utilise les fonctions intrinsèques du compilateur pour les processeurs compatibles Intel MMX/SSE2.

La table 6.1 présente les résultats expérimentaux pour la transposition par la diagonale et par l'antidiagonale d'une image 512×512 dont les éléments sont du type *unsigned integer* de 8 bits sur le processeur Intel Pentium 4 de 2.4 GHz par l'exécution en un seul thread. La zone de mémoire où sont stockées les données est distincte à l'entrée et à la sortie. Nous constatons un gain de temps déjà entre l'implémentation générique qui consiste en l'utilisation des fonctions d'accès au pixel et une implémentation qui utilise le travail avec les pointeurs. Mais le gain que nous obtenons lors de l'utilisation des instructions MMX est plus intéressant, surtout si nous comptons utiliser la transposition comme une des opérations de base dans nos algorithmes de morphologie mathématique.

Ce qui peut être assez surprenant c'est la durée de l'implémentation générique et même celle *via pointer++* pour un tel algorithme de base sur une machine relativement puissante de nos jours et cadencée à 2.4 GHz. Ainsi, nous accueillons avec plaisir la possibilité d'obtenir, sans aucun investissement dans le matériel existant, un algorithme plus rapide.

La figure 6.9 nous montre les représentations graphiques des tests de performance que nous avons effectué pour l'algorithme de la transposition par diagonale et plusieurs tailles d'images. À l'échelle logarithmique, nous verrons bien que la différence entre les implémentations non-SIMD (générique et via pointer++) où nous avons laissé toutes les optimisations au compilateur, et celles qui implémentent notre algorithme SIMD est importante pour toutes les tailles d'images. Avec grands taux d'accélérations s'élevant jusqu'à 33.8 pour les images de 1024×1024 de 8 bits si on compare l'implémentation SIMD utilisant la technologie Intel SSE2 et l'implémentation classique *via pointer++* (cf. tab. 6.1).

Image	Méthode d'implémentation	Transposition par diagonale Temps ms	diagonale Taux d'accélération	antidiagonale Temps ms	antidiagonale Taux d'accélération
$512^2 \times 8$ bits	générique élément par élément	2.61	0.58	3.02	0.50
	via pointer++	1.51	**1.00**	1.51	**1.00**
	instructions MMX	0.30	5.03	0.31	4.87
	instructions SSE2	0.23	6.57	—	—
$1024^2 \times 8$ bits	générique élément par élément	61.3	0.99	61.9	0.99
	via pointer++	60.9	**1.00**	61.7	**1.00**
	instructions MMX	2.2	27.7	2.2	28.0
	instructions SSE2	1.8	33.8	—	—

Implémentation sur Intel Pentium 4 @ 2.4 GHz (single thread, 8 ko L1, 512 ko L2). La zone de mémoire de sortie est distincte de celle d'entrée. Compilateur Intel ICC 8. Taux d'accélération est calculé par rapport à l'implémentation *via pointer++* que nous prenons comme étalon (en gras).

TAB. 6.1 : Algorithmes de transposition par diagonale et antidiagonale ; comparaison des temps de calcul et des taux d'accélération pour diverses implémentations et des tailles d'images

Le deuxième graphique de la même figure, 6.9(b), nous présente encore un comportement intéressant des processeurs sur les chiffres des temps d'exécution normalisés pour 1 pixel. Il s'agit de l'impact de la mémoire cache sur le calcul des images dont la taille excède celle de la mémoire cache. Il s'agit, dans ce cas précis, de la mémoire cache L2 de notre processeur Intel Pentium 4 et dont la taille est de 512 ko.

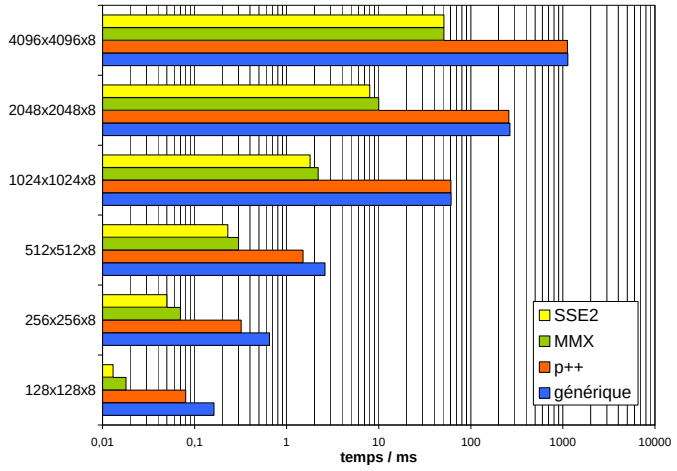
Il y a, en effet, deux points à remarquer. Premièrement, on voit bien que pour les images qui entrent entièrement dans la mémoire cache (images 128^2, 256^2 et 512^2), le coût du calcul est moindre à celui des images qui n'y entrent pas (1024^2, 2048^2, 4096^2). Pour les dernières, nous ne profitons pas d'un accès rapide aux données et le surcoût devrait correspondre au temps d'attente relative à la préparation des données non-présentes dans la mémoire cache.

Deuxièmement, nous pouvons apercevoir un comportement particulier pour les images 1024^2, 2048^2, 4096^2, c'est-à-dire les images dont la taille est plus grande que celle de la mémoire cache L2. Pour ces dernières, l'écart entre les implémentations SIMD et non-SIMD est beaucoup plus important que pour les images qui entrent entièrement dans la mémoire cache. Pourtant, le surcoût des transferts des données entre la mémoire cache et la mémoire principale devrait être, en théorie, le même pour les deux manières d'implémentation, puisque le volume de données transférées est identique.

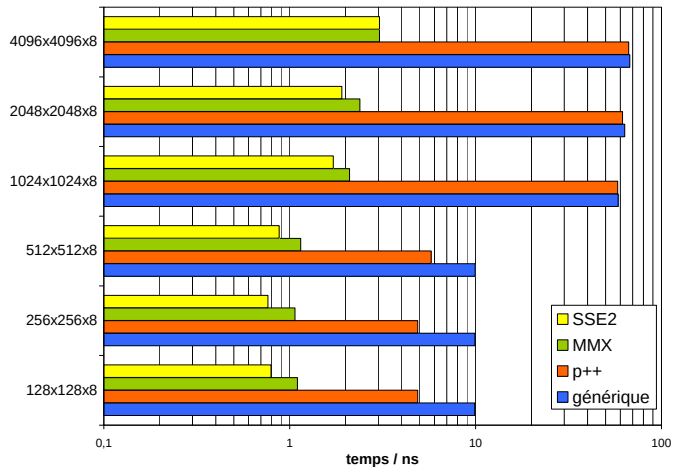
L'explication de ce comportement n'a pas pu être identifiée mais vu que les temps de traitement deviennent importants pour les grandes images, nous n'excluons pas la possibilité que ce comportement soit lié à la manière d'exécution de notre programme dans le système d'exploitation multi-tâche, Linux Man-

```
void inline Transpose8x8_SSE2(
   Iu8vec8 & mm0, Iu8vec8 & mm1,
   Iu8vec8 & mm2, Iu8vec8 & mm3,
   Iu8vec8 & mm4, Iu8vec8 & mm5,
   Iu8vec8 & mm6, Iu8vec8 & mm7
      )
{
      __m128i xmm0, xmm1, xmm2, xmm3,
      __m128i xmm4, xmm5, xmm6, xmm7;

      xmm0 = _mm_movpi64_epi64( (__m64 &) mm0 );
      xmm1 = _mm_movpi64_epi64( (__m64 &) mm1 );
      xmm2 = _mm_movpi64_epi64( (__m64 &) mm2 );
      xmm3 = _mm_movpi64_epi64( (__m64 &) mm3 );
      xmm4 = _mm_movpi64_epi64( (__m64 &) mm4 );
      xmm5 = _mm_movpi64_epi64( (__m64 &) mm5 );
      xmm6 = _mm_movpi64_epi64( (__m64 &) mm6 );
      xmm7 = _mm_movpi64_epi64( (__m64 &) mm7 );

      xmm4 = _mm_unpacklo_epi8(xmm0, xmm4);
      xmm5 = _mm_unpacklo_epi8(xmm1, xmm5);
      xmm6 = _mm_unpacklo_epi8(xmm2, xmm6);
      xmm7 = _mm_unpacklo_epi8(xmm3, xmm7);

      xmm2 = xmm6;
      xmm2 = _mm_unpacklo_epi8(xmm4, xmm2);
      xmm3 = xmm7;
      xmm3 = _mm_unpacklo_epi8(xmm5, xmm3);
      xmm6 = _mm_unpackhi_epi8(xmm4, xmm6);
      xmm7 = _mm_unpackhi_epi8(xmm5, xmm7);
      xmm1 = xmm3;
      xmm1 = _mm_unpacklo_epi8(xmm2, xmm1);
      xmm3 = _mm_unpackhi_epi8(xmm2, xmm3);
      xmm5 = xmm7;
      xmm5 = _mm_unpacklo_epi8(xmm6, xmm5);
      xmm7 = _mm_unpackhi_epi8(xmm6, xmm7);

      (__m64 &)mm0 = _mm_movepi64_pi64(xmm1);
      xmm1 = _mm_srli_si128(xmm1, 8);
      (__m64 &)mm1 = _mm_movepi64_pi64(xmm1);
      (__m64 &)mm2 = _mm_movepi64_pi64(xmm3);
      xmm3 = _mm_srli_si128(xmm3, 8);
      (__m64 &)mm3 = _mm_movepi64_pi64(xmm3);
      (__m64 &)mm4 = _mm_movepi64_pi64(xmm5);
      xmm5 = _mm_srli_si128(xmm5, 8);
      (__m64 &)mm5 = _mm_movepi64_pi64(xmm5);
      (__m64 &)mm6 = _mm_movepi64_pi64(xmm7);
      xmm7 = _mm_srli_si128(xmm7, 8);
      (__m64 &)mm7 = _mm_movepi64_pi64(xmm7);

      _mm_empty();
      return;
}
```

FIG. 6.8 : Code de la transposition par diagonale d'un macro bloc 8×8 écrit manuellement en langage C en utilisant le jeu d'instructions 128 bits Intel SSE2



(a) Temps du calcul



(b) Temps du calcul normalisé pour 1 pixel

FIG. 6.9 : Résultats de diverses implémentations de la transposition par diagonale pour différentes tailles d'images

drake 9.1 dans ce cas-là, à la manière de mesure du temps (plusieurs itérations, temps moyen) ou à un autre phénomène connexe à l'environnement d'exécution.

6.5 Récapitulation, perspectives

Nous avons présenté, dans ce chapitre, quatre opérations possibles pour pouvoir changer le sens de stockage des arrays qui sont notre structure des données de base pour les images. Nous avons présenté également trois approches possibles à leurs implémentations.

Il s'agit de l'algorithme 6.1 qui définit, à travers de la fonction trRot2D, un algorithme travaillant élément par élément et implémentant la définition de ces opérations. Le deuxième algorithme que nous avons présenté était l'algorithme 6.2 qui définit, à travers la fonction trRot2DMB, un algorithme travaillant avec le découpage de l'array d'entrée en macro blocs et qui exécute les opérations localement à l'intérieur de chacun de ceux-ci élément par élément mais qui exécute la même opération également à l'échelle des macro blocs. Le troisième algorithme, l'algorithme 6.6, qui définit, à travers la fonction trRot2DMBSIMD, un algorithme qui travaille en utilisant les macro blocs mais qui emploie à l'intérieur de ceux-ci les instructions spécialisées des architectures multimédia – les fonctions shuffle.

Le dernier algorithme sera le plus utile dans les applications morphologiques que nous allons décrire par la suite car il réduit le temps nécessaire pour le changement de stockage des images. Les résultats expérimentaux présentés dans la tab. 6.1 démontrent bien son utilité par rapport aux implémentations naïves qui travaillent par la définition et implémentent, en effet, l'algorithme trivial 6.1.

Dans nos explications pour 2D, nous nous sommes spécifiquement concentrés sur les macro blocs de taille 8×8. Cela pour la bonne raison que les architectures des ordinateurs qui s'installent sur le marché grand public sont, en effet, les architectures de 64 bits (Intel IA-64, AMD64, SuperH SHmedia) et nous pouvons voir directement les aspects applicatifs de nos algorithmes pour ces architectures. Pour démontrer que ce sujet est d'un intérêt majeur pour les application, nous pouvons citer les articles[Lee00, LFB01] qui traitent d'un sujet connexe (ils sont focalisés sur l'architecture Intel IA-64) mais qui ne sont pas directement orientés vers un changement du sens de stockage des image pour un traitement SIMD.

L'approche macro bloc élément par élément peut également trouver son emploi lors d'un travail avec des images plus grandes que les mémoires caches de l'architecture cible. Dans ce cas précis, il serait avantageux pour un meilleur emploi de la mémoire cache de travailler par macro blocs et de combiner, sur les architectures multimédia, l'approche de découpage en macro blocs plus grands que la taille des registres avec l'approche SIMD à l'intérieur des registres.

Cette page est blanche par intention

CHAPITRE 7

Algorithmes de voisinage dépendant du sens prédéfini de parcours de l'image

Les algorithmes travaillant sur le voisinage et dont le traitement dépend du sens de parcours de l'image forment un autre groupe d'algorithmes de la morphologie mathématique. En effet, nous pouvons y classer tous les algorithmes qui utilisent la propagation d'une valeur dans un sens défini. De ce point de vue, les algorithmes les plus naturels de ce travail sont ceux qui calculent la *fonction distance*[1].

Dans les applications destinées au traitement en temps réel, nous nous intéressons aux fonctions distance qui peuvent nous rendre une approximation de la distance le plus rapidement possible. De ce point de vue, nos cibles prioritaires seront les algorithmes des fonctions distances non-euclidiennes. Ces algorithmes, comme on le verra par la suite, utilisent des techniques particulières pour le traitement et elles sont transposables également au traitement SIMD des images sur les architectures multimédia.

L'approche que l'on va décrire ne se restreindra pas seulement aux fonctions distances. D'autres types d'algorithmes peuvent être implémentés suivant la même approche. Les opérations morphologiques qui peuvent en bénéficier sont représentées par la *reconstruction* morphologique (cf. livre[Soi03] de référence) et par tous les algorithmes dérivées de cette dernière. Nous nous consacrerons dans ce chapitre plus particulièrement aux *nivellements* qui sont des filtres morphologiques d'une importance cruciale pour les applications de filtrage et de segmentation d'images.

7.1 Particularité du sens du parcours pour le traitement SIMD du voisinage

Dans les traitements qui n'utilisent pas l'approche vectorielle et, par conséquent, n'exploitent pas le parallélisme des données à l'échelle d'un registre, les sens du parcours plus que classiques sont ceux qui parcourent l'image en sens vidéo et anti-vidéo, bien connus des algorithmes d'évaluation de la fonction distance *chamfer*. La figure 7.1 illustre cette situation.

Le calcul standard des fonctions distance approximatives est assuré par les méthodes qui décomposent le kernel en deux parties et le calcul en deux parcours (vidéo et anti-vidéo) et sont appelées les distances *chamfer*. Formellement, nous pouvons décrire les parcours complets par la composition de deux fonctions p_1 et p_2 :

$$\text{parcours}\ \ ar = p_1 \circ p_2\ \$\ ar$$

Ces types de parcours nous offrent la propagation des valeurs, à l'échelle des pixels, en deux axes en même temps. Donc, l'avantage de cette approche pour le calcul de la fonction distance est, sans aucun doute, dans l'utilisation de seulement deux parcours de l'image entière.

[1] Nous renvoyons le lecteur à la thèse[Cui99] d'Olivier Cuisenaire pour plus de détails théoriques sur les fonctions distance.

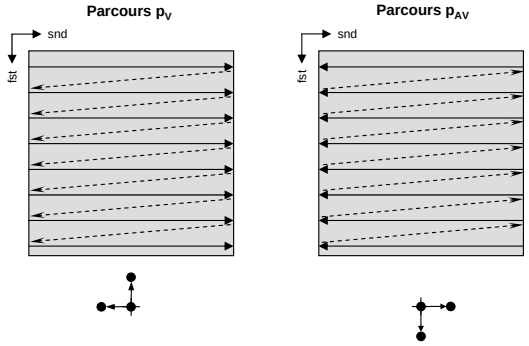


FIG. 7.1 : Décomposition du kernel en deux parties et en deux parcours de l'image lors de l'évaluation des fonctions distance *chamfer*. L'exemple d'élément structurant pour le 4-voisinage et la grille carrée.

De plus, certaines architectures matérielles dédiées à la morphologie mathématique implémentent ces algorithmes en utilisant le stockage local de plusieurs lignes vidéo, on parle des *lignes à retard*. Les éléments du voisinage local d'une partie du kernel décomposé est extrait à partir de ces lignes par des mécanismes standards de l'extraction du voisinage.

Ayant à disposition des architectures avec les capacités SIMD, nous nous demandons comment profiter du parallélisme de données pour ces traitements. Et nous nous apercevons que le fait d'utiliser les parcours vidéo et anti-vidéo, qui étaient présentés comme avantageux pour le traitement à l'échelle des pixels, devient gênant pour les traitements à l'échelles des vecteurs paquetés. Il est, en effet, extrêmement coûteux d'effectuer la propagation d'une valeur élément par élément à l'intérieur d'un vecteur paqueté, surtout à cause du non-support d'un tel traitement par les jeux d'instructions multimédia[1].

Pourtant, le traitement à l'échelle des vecteurs paquetés peut utiliser la force du calcul SIMD. C'est pourquoi nous n'allons pas utiliser le parcours vidéo ou anti-vidéo directement mais nous les diviserons en quatre phases en total. Dans chacune des phases, nous allons utiliser une direction de propagation différente et nous allons travailler à l'échelle des vecteurs paquetés. La figure 7.2 illustre cette situation.

Nous pouvons y percevoir la différence entre le traitement à l'échelle des éléments de base, q.v. fig. 7.2(a), et le traitement à l'échelle des vecteurs paquetés, q.v. fig. 7.2(b). La dernière figure montre également de quelle manière on regroupe les éléments de l'image dans les vecteurs paquetés. Notons que l'axe de vectorisation est, dans les quatre phases, perpendiculaire au sens du parcours. Le kernel du calcul est également décomposé en quatre parties, chacune d'elles à utiliser dans une phase différente. La formule suivante

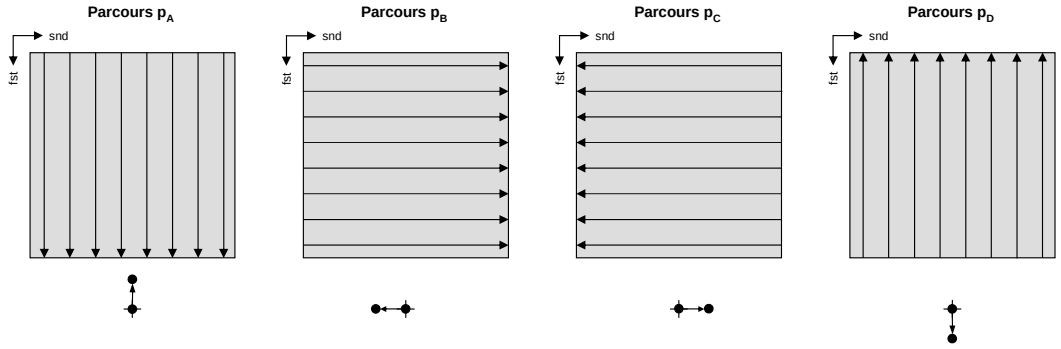
$$\mathsf{parcours}\ \ ar = p_D \circ p_C \circ p_B \circ p_A\ \$\ ar$$

décrit formellement ce procédé. Notons que l'ordre d'application de ces phases, comme présenté par la formule précédente, n'est qu'une possibilité parmi d'autres. Il existe, en effet, quatre manières d'ordonner les phases et nous pouvons les utiliser dans les algorithmes ayant la même structure de fonctionnement que la fonction distance :

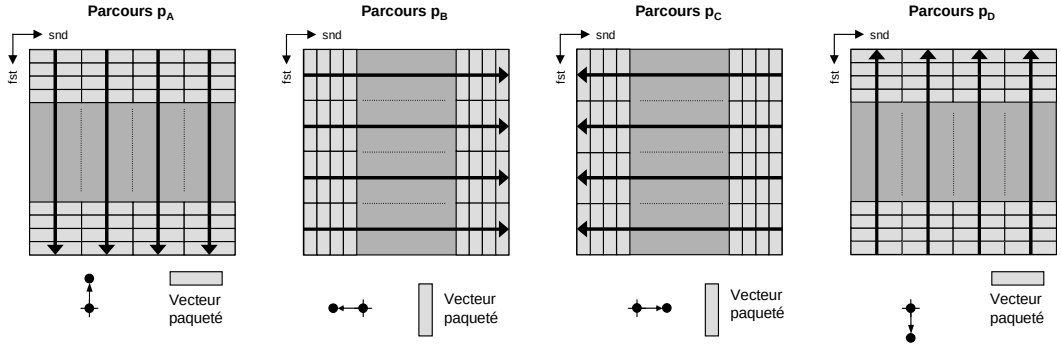
$$\begin{aligned}\mathsf{parcours}\ \ ar &= (p_D \circ p_C) \circ (p_B \circ p_A)\ \$\ ar \\ &= (p_D \circ p_C) \circ (p_A \circ p_B)\ \$\ ar \\ &= (p_C \circ p_D) \circ (p_B \circ p_A)\ \$\ ar \\ &= (p_D \circ p_C) \circ (p_A \circ p_B)\ \$\ ar\end{aligned}$$

La problématique de la vectorisation a déjà été discutée dans la section dédiée au paquetage et dépaquetage des données (cf. 4.4.3, page 68). En pratique, l'axe de vectorisation est choisi comme identique à celui de stockage des données dans la mémoire. Ce qui veut dire que deux (p_A et p_D) des quatre phases de la propagation SIMD sont applicables directement comme présenté ci-dessus. Pour pouvoir appliquer l'approche SIMD dans les deux phases restantes (p_B et p_C), nous devons faire appel aux techniques de

[1] Nous nous basons sur les jeux d'instructions que nous avons pu consulter – Intel IA-32 (MMX, SSE, SSE2, SSE3) et SuperH SHmedia



(a) Travail avec les éléments de base



(b) Travail SIMD avec les vecteurs paquetés

FIG. 7.2 : Décomposition du kernel en quatre parties et en quatre parcours de l'image. L'exemple d'élément structurant pour le 4-voisinage et la grille carrée.

changement de l'axe de stockage des données. Il s'agit des techniques présentées dans le chapitre 6, page 127, et nous allons utiliser la *transposition par diagonale* en particulier.

Ceci dit, les deux phases de propagation (p_B et p_C) dont l'orientation est parallèle à l'axe de stockage des données dans la mémoire (l'axe snd dans ce cas) vont faire appel à la transposition de l'image dans la mémoire. Après cette transposition, les données avec lesquelles nous voulons travailler seront prêtes pour le traitement par les instructions SIMD. En même temps, le sens du parcours que nous devons utiliser lors de travail avec ces données transposées ne sera pas celui appliqué aux données originaires, il doit également être transposé. Après l'application d'un kernel, nous devons faire appel à une deuxième transposition pour obtenir les données orientées dans le bon sens. La fig. 7.3 illustre cette idée sur l'exemple de la phase p_B pour laquelle les données, après être transposées, utilisent le même sens de parcours que celui de la phase p_A. Formellement, nous pouvons décrire ce procédé par la formule suivante :

$$p_B \ \$ \ ar = t_D \circ p_A \circ t_D \ \$ \ ar$$

Suivant la même idée, nous pouvons dériver également la formule pour le calcul de la phase p_C qui utilisera pour les données transposées, le même sens de parcours que la phase p_D

$$p_C \ \$ \ ar = t_D \circ p_D \circ t_D \ \$ \ ar$$

Si nous assemblons toutes ces idées, nous pourrons construire un schéma global de fonctionnement qui sera utilisé lors du travail avec les données paquetées. Formellement, nous pouvons écrire :

$$\begin{aligned} \text{parcours } ar &= p_D \circ p_C \circ p_B \circ p_A \ \$ \ ar \\ &= p_D \circ (t_D \circ p_D \circ t_D) \circ (t_D \circ p_A \circ t_D) \circ p_A \ \$ \ ar \\ &= p_D \circ t_D \circ p_D \circ (t_D \circ t_D) \circ p_A \circ t_D \circ p_A \ \$ \ ar \\ &= p_D \circ t_D \circ p_D \circ p_A \circ t_D \circ p_A \ \$ \ ar \end{aligned}$$

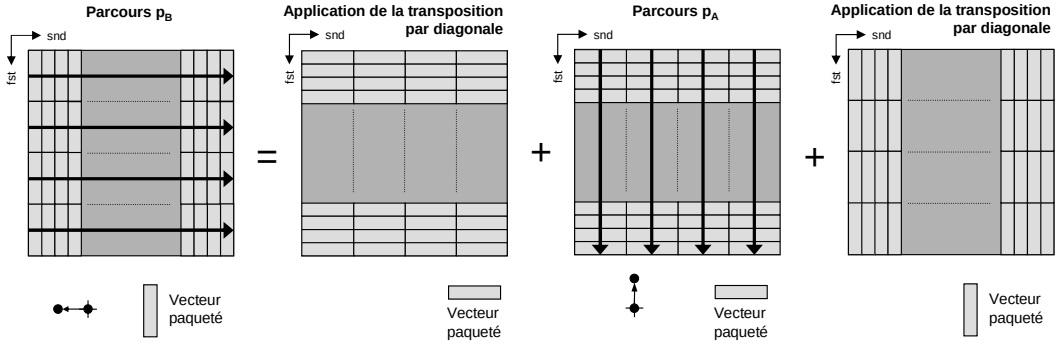


FIG. 7.3 : Remplacement de la propagation SIMD en direction parallèle au sens du stockage par les transpositions par diagonale de l'array et par l'application de la propagation en sens perpendiculaire à l'axe de vectorisation. L'exemple d'élément structurant pour la grille carrée et le 4-voisinage.

où la dernière ligne présente le schéma après l'élimination de la double transposition qui est redondante. Dans le cas où les données sont paquetées d'une façon différente de celle présentée sur la fig. 7.3, les formules analogiques peuvent être construites en reflétant ce sens de paquetage particulier.

7.2 Skeletons applico-réductifs pour la propagation

Pour pouvoir exprimer formellement la manière dont on travaillera lors des propagations, nous définissons deux skeletons : mfoldl et mfoldl1. Nous les avons nommées les skeletons *applico-réductifs* car ils combinent deux opérations de base, l'*application* et la *réduction* dans leurs corps. Leurs fonctionnements sont très proches de deux skeletons du Haskell : **scanl** et **scanl1**, mais ils diffèrent par certains détails et leurs définitions ne sont pas exactement les mêmes.

Le skeleton mfoldl :

```
mfoldl  ::  (α → β → α) → α → [β] → [α]
mfoldl _ _ []     = []
mfoldl f v (e:es) = r : (mfoldl f r es)    where r = f v e
```

prend trois arguments. Le premier est la fonction f de deux arguments qui assure la fonctionnalité d'*application*. La fonction f est appliquée sur l'argument v de ce skeleton et sur le premier élément e de la liste $(e : es)$. Son résultat est inscrit dans le stream résultant mais également réutilisé comme argument dans l'application suivante de la fonction f sur le deuxième élément du stream d'entrée. Cette propagation se poursuit jusqu'à ce que le stream d'entrée soit vide. La figure 7.4 illustre graphiquement ce fonctionnement. Notons que ce skeleton retourne un stream qui est le résultat de l'application de la fonction f. Le résultat de la *réduction* n'est pas retourné explicitement mais incorporé dans le stream résultant. C'est, en effet, le dernier élément de ce stream qui porte la valeur résultante de la réduction.

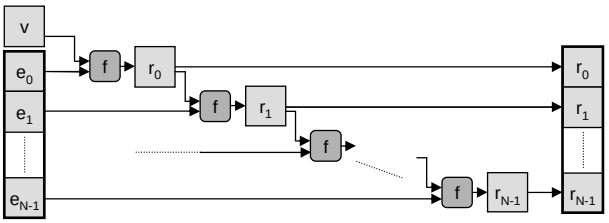


FIG. 7.4 : Fonctionnement du skeleton applico-réductif mfoldl.

Dans certains cas, il sera plus pratique d'utiliser le skeleton algorithmique mfoldl1 qui assure la fonctionnalité applico-réductive sur le stream. Par rapport au skeleton mfoldl décrit précédemment, nous ne passons que la fonction f et le stream d'entrée $(e : es)$ comme arguments :

```
mfoldl1  ::  (α → α → α) → [α] → [α]
mfoldl1 _ []     = []
mfoldl1 f (e:es) = e : (mfoldl f e es)
```

Ce skeleton n'applique pas la fonction *f* sur le premier élément *e* du stream et le retourne aussitôt dans le stream résultant. En revanche, sa valeur est utilisée comme argument d'entrée pour la réduction qui est assurée, pour tous les éléments suivants *es* du stream, par l'application de la fonction mfoldl. La figure 7.5 illustre cette situation.

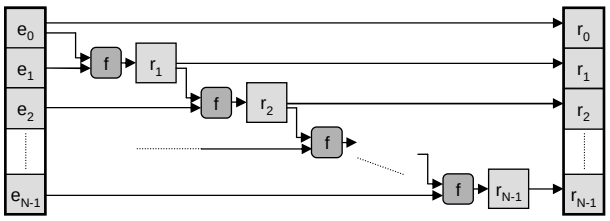


FIG. 7.5 : Fonctionnement du skeleton applico-réductif mfoldl1.

7.3 Skeleton algorithmique de la propagation SIMD en 4-voisinage

Après avoir expliqué la particularité du sens du parcours pour les propagations dans le cas où on travaille avec les vecteurs paquetés et après avoir expliqué le principe de fonctionnement des algorithmes qui s'appuient sur les techniques de changement de l'axe de stockage de données, nous sommes prêts à construire le premier skeleton algorithmique qui utilisera ce type de propagation.

Il est évident qu'en utilisant les parcours de l'image pour la propagation, comme illustrés sur la fig. 7.3, nous utiliserons le découpage de l'array en macro blocs. Un macro bloc sera constitué par les données qui sont parcourues lors de la propagation. Ainsi, la propagation s'effectuera à l'échelle des macro blocs et les évaluations à l'intérieur de chacun des macro blocs seront indépendantes les unes des autres.

7.3.1 Propagation à l'intérieur d'un macro bloc

Expliquons alors le fonctionnement d'une telle propagation à l'intérieur d'un macro bloc. Par la suite, nous allons bâtir notre algorithme entier à partir de ce fonctionnement de base.

Le skeleton pGenMB définit la propagation dans un macro bloc, dont les éléments sont les vecteurs paquetés, à l'aide de la fonction *f* qui définit l'opération exacte à effectuer. Il est naturel que dans le cœur de ce skeleton, nous nous appuyons sur un des skeletons applico-réductifs ; mfoldl1 dans ce cas précis. La définition que nous présentons ici ne fait, en effet, que concrétiser l'utilisation de ce dernier pour une situation qui nous intéresse – la propagation à l'intérieur d'un macro bloc :

Tout d'abord nous construisons le stream des index (ligne 10) en utilisant la fonction streamAr2D avec les paramètres *how*, qui définit le sens du parcours, et *mb*, qui est un array des vecteurs paquetés et qui représente le macro bloc. Notre algorithme commence par l'utilisation de ce stream d'index *ixs* (ligne 8). Sur chaque index de ce stream, nous appliquons (ligne 7) la fonction extraction des éléments à partir du macro bloc (*mb*!). Notons que dans ce cas, nous ne traitons pas les bords de l'image et, par conséquent, nous n'avons pas besoin d'assurer l'extraction des éléments par une fonction particulière qui gérerait les effets de bord. Ainsi, nous obtenons le stream des vecteurs paquetés sur lequel nous appliquons (ligne 6) directement le skeleton applico-réductif mfoldl1 avec la fonction *f* comme argument qui assure la propagation. Les expressions restantes (lignes 5 et ligne 4) qui utilisent les fonctions **zip** et **array** constituent la manière standard de créer un macro bloc de sortie à partir des résultats de la propagation.

Algorithme 7.1 : pGenMB, skeleton algorithmique de la propagation SIMD à l'intérieur d'un macro bloc composé des vecteurs paquetés pour le 4-voisinage et la grille carrée

```
1  pGenMB   ::  (Ord α) ⇒ [Char] → (PVec I α → PVec I α → (PVec I α, PVec I α))
2               → Ar (I,I) (PVec I α) → Ar (I,I) (PVec I α)
3  pGenMB how f mb =
4                    (array (bounds $ mb))
5                  ∘ (zip ixs)
6                  ∘ (mfoldl1 f)
7                  ∘ (map ( mb! ))
8                  $ ixs
9      where
10         ixs = streamAr2D how mb
```

7.3.2 Phase généralisée de la propagation

Montrons maintenant la structure d'une phase de propagation entière qui utilisera le skeleton pGenMB pour les macro blocs que l'on vient de définir. C'est la fonction pGen, définie par l'algorithme 7.2, qui présente la description formelle d'une telle phase de propagation sur l'image entière. Son fonctionnement est assez simple, elle ne fait que vectoriser (ligne 10) l'array d'entrée ar, applique ensuite (ligne 9) le découpage de cet array aux $m \times n$ macro blocs et crée à partir d'un array des macro blocs le stream des macro blocs (ligne 8). Sur la ligne 7, on applique à chaque macro bloc de ce stream la fonction de la propagation à l'intérieur d'un macro bloc pGenMB. Les expressions sur les lignes restantes ne font que reconstruire, à partir d'un stream des macro blocs résultant, l'array des éléments de base par l'approche inverse. Nous transformons (ligne 6) le stream des macro blocs en un array des macro blocs. À partir de ce dernier, nous construisons (ligne 5) un array des vecteurs paquetés et nous procédons à la dévectorisation (ligne 4) pour obtenir l'array résultat dont les éléments sont les éléments de base du type α

Algorithme 7.2 : pGen, skeleton algorithmique général d'une phase de propagation SIMD pour le 4-voisinage et la grille carrée

```
pGen   ::  (Ord α) ⇒ [Char] → [Char] → I → (PVec I α → PVec I α → (PVec I α, PVec I α))
       → Ar (I,I) α → Ar (I,I) α
pGen fworbw axe pvecsz f ar =
                    (mkAr2DFromAr2DPVec axe)
                  ∘ arrayFromMxNBlocs
                  ∘ (listArray ((1,1),(m,n)))
                  ∘ (map (pGenMB how f ))
                  ∘ elems
                  ∘ (arrayToMxNBlocs m n)
                  ∘ (mkAr2DPVec axe pvecsz)
                  $ ar
    where
      how = if (axe == "Fst") then (fworbw ++ "Snd") else (fworbw ++ "Fst")
      (p,q) = dimsAr2D ar
      (m,n) = if (axe == "Fst") then (div p pvecsz, 1) else (1, div q pvecsz)
```

Ce skeleton algorithmique est général et commun pour les deux sens de parcours que nous allons employer ("Fwd" et "Bwd") et qui sont précisés par l'argument *fworbw*. Le deuxième argument de ce skeleton, *axe*, nous définit l'axe de stockage des données dans la mémoire ("Fst" ou "Snd") et *pvecsz* définit la taille du vecteur paqueté que nous voulons utiliser et qui correspond au nombre d'éléments qui

peuvent être traités par les instructions SIMD de notre architecture multimédia en même temps. *f* est la fonction qui définit l'opération de propagation (notons qu'elle est définie pour les vecteurs paquetés) et *ar* est l'array d'entrée.

7.3.3 Propagations SIMD sur l'image entière pour le 4-voisinage et la grille carrée

Ainsi, nous avons présenté les outils de base pour pouvoir finalement définir le skeleton propAlgSQR4 de la propagation SIMD pour le 4-voisinage et la grille carrée. C'est l'algorithme 7.3 qui définit ce skeleton.

Algorithme 7.3 : propAlgSQR4, skeleton algorithmique de la propagation SIMD pour la grille carrée et le 4-voisinage

```
propAlgSQR4 :: (Ord α) ⇒ [Char] → I → (PVec I α → PVec I α → (PVec I α, PVec I α))
               → Ar (I,I) α → Ar (I,I) α
propAlgSQR4 axe pvecsz f ar = p2 ∘ tD ∘ p2 ∘ p1 ∘ tD ∘ p1 $ ar
  where
    p1 = pGen "FW" axe pvecsz f
    p2 = pGen "BW" axe pvecsz f
    tD = trRot2DMBSIMD "TD" axe pvecsz
```

Nous voyons que sa définition est beaucoup plus claire que celles des skeletons précédents. En effet, nous avons caché tout le travail lourd dans la fonction de parcours pGen et nous obtenons une structure définissant le strict nécessaire pour la description de ce type de propagation.

Nous y reconnaissons, sur la ligne 3, l'enchaînement des opérations :

```
p2 ∘ tD ∘ p2 ∘ p1 ∘ tD ∘ p1 $ ar
```

que nous avons déjà mentionné dans la section 7.1, page 7.3 et qui n'utilise que 2 sens de parcours principaux, appliquant ainsi l'approche de la transposition pour contourner les propagations parallèles à l'axe de stockage des données dans la mémoire, comme illustrées sur la fig. 7.3, page 7.3. Mais nous utilisons l'approche plus générale qui travaille avec les phases de propagation p_1 et p_2, qui ne sont concrétisées qu'à l'intérieur de la phase de propagation pGen selon la valeur de l'axe de stockage transmise par le paramètre *axe*. Le paramètre *pvecsz* définit le nombre d'éléments qui peuvent être traités en parallèle par les instructions SIMD de notre architecture et *f* définit l'opération à effectuer lors de la propagation ; *ar* est l'array d'entrée.

Remarquons que pour changer l'axe de stockage de données, nous utilisons la fonction généralisée de la transposition /rotation d'un macro bloc trRot2DMBSIMD que nous avons définie dans le chapitre 6 dédié à ce sujet. Rappelons que cette fonction est concrétisée par le paramètre ″TD″ pour effectuer la transposition par diagonale et qu'elle utilise l'approche des macro blocs lors de son évaluation. La taille d'un macro bloc est définie par la taille du vecteur paqueté *pvecsz*.

7.3.4 Calcul de la fonction distance

Les skeletons que nous venons de présenter ne définissent que la structure du traitement. La fonctionnalité concrète va utiliser ces skeletons pour les spécialiser et leur donner la forme finale de l'algorithme qui exécute l'opération souhaitée. Dans notre cas, l'opération que nous ciblons tout d'abord est la fonction distance *chamfer* calculée sur le 4-voisinage et la grille carrée.

Pour pouvoir la définir, nous devons préciser certains paramètres dépendant de l'application concrète. Il s'agit notamment de l'axe de stockage des données dans la mémoire *axe* et de la taille du vecteur paqueté *pvecsz*.

En ce qui concerne les données d'entrée, nous devons avoir deux arrays : le premier avec l'image du masque, qui est une image binaire et où les valeurs ″True″ définissent la zone dans laquelle nous calculons

la fonction distance, et les valeurs ″False″ dans le cas contraire. Et puis, il nous faut un deuxième array qui contiendra les résultats de la distance et dont le type de stockage serait suffisant pour contenir les valeurs calculées. Ces valeurs doivent être initialisées avant le début de l'évaluation à 0 dans la zone désignée par les valeurs ″False″ du masque et à la valeur la plus grande possible ($+\infty$ ou la valeur maximale du type de stockage) dans la zone désignée par les valeurs ″True″ du masque. La figure 7.6 illustre cette situation.

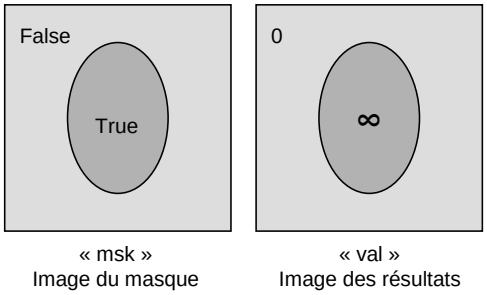


FIG. 7.6 : Initialisation des images avant l'exécution de l'algorithme de la fonction distance.

Pour pouvoir exprimer le fonctionnement d'une manière simple, il est utile d'associer dans un seul élément le masque et la valeur calculée de la fonction distance. Ceci nous permettra de travailler avec un seul array dont les éléments sont des tuples (masque,résultat). L'algorithme de la fonction distance aura ensuite la forme :

```
fncDistanceSQR4  ::  [Char] → I → Ar (I,I) (Bool, α) → Ar (I,I) (Bool, α)
fncDistanceSQR4 axe pvecsz ar = propAlgSQR4 axe pvecsz
                    ( λ (vmsk,vval) (emsk,eval) → (emsk, cndmoveSIMD (testSIMD emsk (==) True)
                                                                     (minSIMD (vval+1) eval)
                                                                     (eval)
                    )
                    ar
```

où l'expression λ définit l'opération de propagation. Dans ce cas précis, il s'agit de la fonction distance directionnelle avec une distance entre les pixels égale à 1. $(vmsk, vval)$ définit l'élément voisin et $(emsk, eval)$ définit l'élément central pour lequel nous effectuons l'évaluation, les deux exprimés par un tuple (masque, valeur).

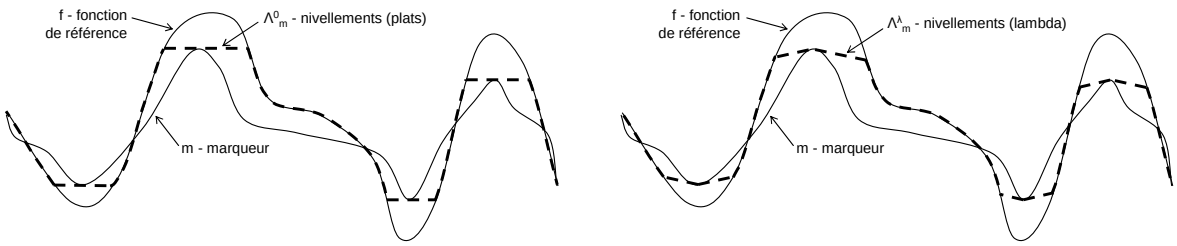
7.3.5 Calcul des nivellements

7.3.5.1 Nivellements

Outre la fonction distance, le style de travail que nous avons décrit par les skeletons algorithmiques de la propagation peut être utilisé également pour définir les algorithmes évaluant des nivellements[Mey03], cela non seulement pour les nivellements plats, cf. fig. 7.7(a), mais également pour les lambda-nivellements, cf. fig. 7.7(b). Ainsi présentés, nous pouvons percevoir les nivellements plats en tant que cas spécial des lambda-nivellements pour la valeur $\lambda = 0$.

Notons également que la technique de propagation des valeurs lors du travail avec les nivellements est très semblable à celle que nous utilisons lors du travail avec la fonction distance, même si les nivellement constituent une opération plus complexe que la fonction distance.

En revanche, ils diffèrent de la fonction distance par leur caractère géodésique et par la propagation des valeurs jusqu'à l'idempotence. Par la suite, nous n'allons présenter que les définitions des algorithmes pour une seule itération, l'application répétitive de ces algorithmes peut être facilement dérivée suivant les même règles que celles présentées pour les opérations géodésiques non-dépendantes du sens de parcours, cf. la section 5.3, page 115.



(a) Nivellements plats *(b)* λ-nivellements

FIG. 7.7 : Nivellements

7.3.5.2 Nivellements en tant que combinaison des nivellements partiels

L'implémentation des nivellements que nous allons présenter va utiliser les skeletons que nous avons décrits précédemment et va s'appuyer sur deux nivellements partiels que nous appelons les sur-nivellements et les sous-nivellements et qui sont illustrés sur la fig. 7.8. Ces deux opérations ont, en effet, la structure de fonctionnement de la propagation identique, ce qui les différencie est l'opération du kernel d'exécution.

Ces fonctions ont des propriétés très intéressantes. Étant donné la fonction de référence f et le marqueur m, les lambda-sous-nivellements $\Lambda_m^{\lambda-}(f)$ de la fonction f contraintes par le marqueur m sont identiques aux nivellements $\Lambda_m^{\lambda}(f)$ dans le domaine de l'image où $m > f$, dans le reste du domaine ils sont identiques à la fonction f. Par dualité, les lambda-sur-nivellements $\Lambda_m^{\lambda+}(f)$ de la fonction f contraintes par le marqueur m sont identiques aux nivellements $\Lambda_m^{\lambda}(f)$ dans le domaine de l'image où $m < f$, dans le reste du domaine ils sont identiques à la fonction f.

Afin d'obtenir les lambda-nivelements $\Lambda_m^{\lambda}(f)$ à partir de ces sur- et sous-nivellements, nous allons combiner ces deux derniers selon la formule suivante :

$$\Lambda_m^{\lambda}(f) = \begin{cases} \Lambda_m^{\lambda-}(f) & m > f \\ f & m = f \\ \Lambda_m^{\lambda+}(f) & m < f \end{cases} \tag{7.1}$$

L'équation 7.1 est la définition mathématique théorique. Dans la pratique nous allons profiter des propriétés des nivellements pour diminuer le nombre d'opérations arithmétiques à effectuer. Tout d'abord, nous allons profiter de l'identité des nivellements avec la fonction de référence f pour $m = f$ afin d'éliminer une condition à vérifier. Les sur- et sous-nivellements sont bornés sur les sous-domaines et les expressions suivantes sont valides :

$$m \geq \Lambda_m^{\lambda-}(f) \geq f, \quad \forall m \geq f m \leq \Lambda_m^{\lambda-}(f) \leq f, \quad \forall m \leq f \tag{7.2}$$

et nous allons les utiliser pour éliminer la fonction f de la formule pour la combinaison des nivellements ce qui va se traduire par le travail avec une image en moins. Ainsi, nous pouvons décrire le même travail par la formule suivante :

$$\Lambda_m^{\lambda}(f) = \begin{cases} \Lambda_m^{\lambda-}(f) & m \geq \Lambda_m^{\lambda-}(f) \\ \Lambda_m^{\lambda+}(f) & m < \Lambda_m^{\lambda-}(f) \end{cases} \tag{7.3}$$

C'est l'équation 7.3 que nous allons utiliser en pratique pour effectuer la combinaison des deux nivellements partiels.

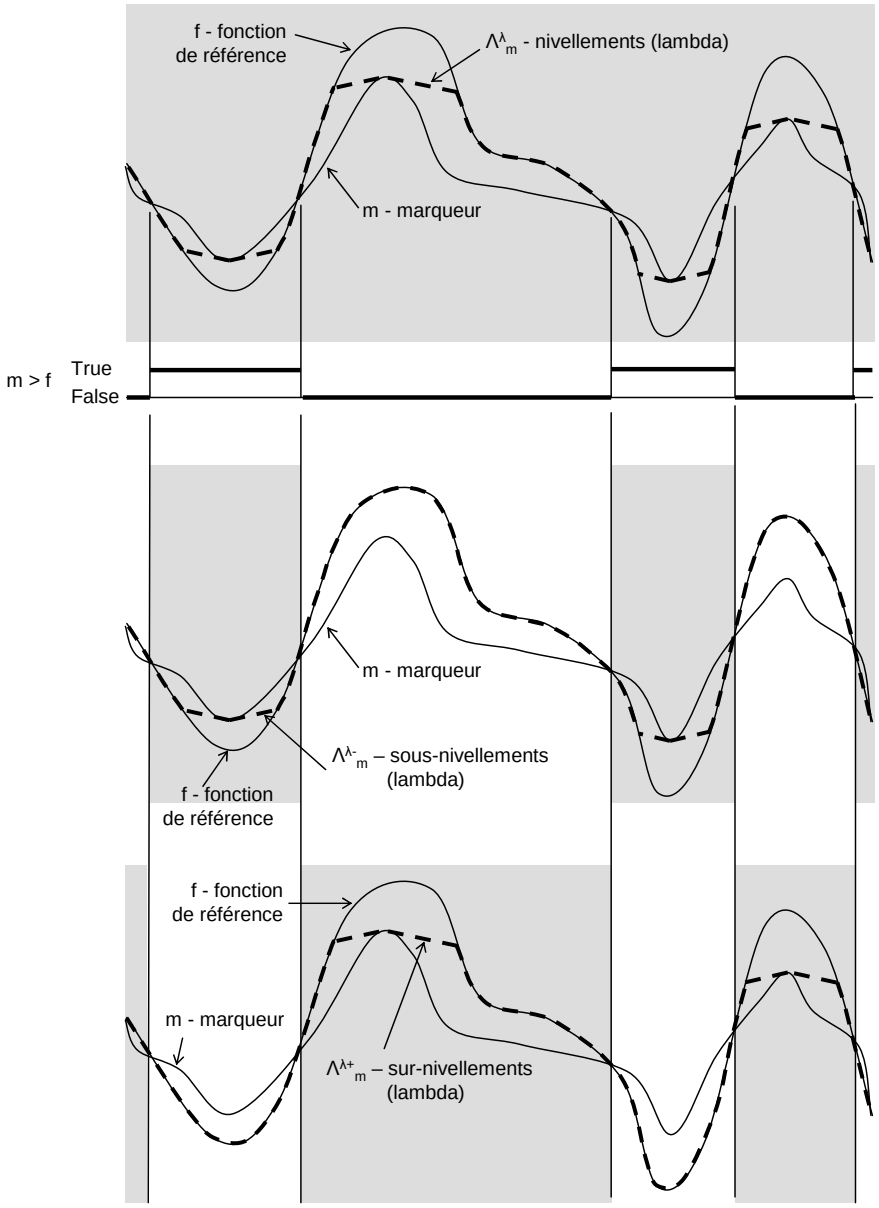


FIG. 7.8 : Calcul des nivellements en tant que combinaison des sur-nivellements et les sous-nivellements

7.3.5.3 Algorithmes pour une itération des nivellements partiels

Le skeleton de la propagation SIMD que nous avons défini dans la section 7.3.3 va nous servir de charpente pour la construction des algorithmes qui vont implémenter les sur-nivellements et les sous-nivellements. Tout d'abord, clarifions les images que nous attendons à l'entrée de nos algorithmes et quelle sera leur fonction lors de la propagation.

La fonction de référence *f* (cf. fig. 7.8) va passer dans les fonctions de propagation en tant que masque et les valeurs du marqueur *m* des nivellements (cf. fig. 7.8) seront utilisées en tant que valeurs initiales à partir desquelles nous commencerons la propagation et qui vont devenir, après avoir appliqué les quatre phases de propagation SIMD, les valeurs résultantes d'une itération des nivellements partiels. La figure 7.9 illustre cette situation.

L'array d'entrée *ar* des fonction définissant les nivellements partiels va être composé de la même manière que celui utilisé par la fonction distance – par des tuples (masque, valeur) et le type de cet array sera, par conséquent, Ar (I, I) (α, α).

(a) Image du masque "*msk" pour la propagation correspondant à la fonction *f* de référence des nivellements

(b) Image des valeurs à propager "*val" correspondant à la fonction *m* du marquer des nivellements

FIG. 7.9 : Les images d'entrée aux fonctions de sur- et sous-nivellements et l'exemple de leur contenu.

Une itération de lambda-sous-nivellements est définie par la fonction lolevelSQR4 :

```
lolevelSQR4   ::  [Char] → I → I → Ar (I,I) (α,α) → Ar (I,I) (α,α)
lolevelSQR4  axe pvecsz lmb ar = propAlgSQR4 axe pvecsz
                ( λ (vmsk,vval) (emsk,eval) → maxSIMD emsk (minSIMD (vval+lmb) eval) )
                ar
```

où le terme λ définit le kernel de la propagation directionnelle et le paramètre *lmb* définit la pente des lambda-sous-nivellements. En spécifiant sa valeur à 0, nous obtenons les sous-nivellements plats. La signification des autres paramètres de cette fonction est identique à celle des paramètres pour la fonction distance (cf. 7.3.4) : *axe* définit l'axe de stockage des données dans la mémoire, *pvecsz* est le nombre des éléments que nous pouvons traiter en même temps par les instructions SIMD de notre architecture multimédia et *ar* est l'array d'entrée composé des tuples (masque, valeur).
Par analogie, nous définissons une itération des lambda-sur-nivellements par la fonction hilevelSQR4 :

```
hilevelSQR4   ::  [Char] → I → I → Ar (I,I) (α,α)→ Ar (I,I) (α,α)
hilevelSQR4  axe pvecsz lmb ar = propAlgSQR4 axe pvecsz
                ( λ (vmsk,vval) (emsk,eval) → minSIMD emsk (maxSIMD (vval−lmb) eval) )
                ar
```

où le seul changement par rapport à la fonction des lambda-sous-nivellements est dans la définition du terme λ définissant le kernel de la propagation directionnelle.

7.3.5.4 Algorithme pour une itération des nivellements

Comme nous l'avons déjà expliqué dans la section 7.3.5.2, une itération des nivellements (finaux) sera obtenue par la combinaison des résultats d'une itération des sur- et sous-nivellements (partiels) selon l'équation 7.3.
La fonction levelSQR4 définit cette composition :

```
levelSQR4   ::  [Char] → I → I → Ar (I,I) (α,α)→ Ar (I,I) (α,α)
levelSQR4 axe pvecsz lmb ar =
        (mkAr2DFromAr2DPVec axe)
      ∘ listArray (bounds $ lo)
      map2 ( λ (lomsk,loval) (himsk,hival) → cndmoveSIMD (testSIMD lomsk (≥) loval) loval hival )
        (elems∘(mkAr2DPVec axe pvecsz)$ lo)
        (elems∘(mkAr2DPVec axe pvecsz)$ hi)
  where
    lo = (lolevelSQR4 axe pvecsz lmb)$ ar
    hi = (hilevelSQR4 axe pvecsz lmb)$ ar
```

Nous y évaluons d'abord les résultats *lo* et *hi* de nivellements partiels en utilisant la fonction lolevelSQR4 des sous-nivellements et la fonction hilevelSQR4 des sur-nivellements. L'expression λ définit sur place la fonction de composition SIMD des nivellements en utilisant la fonction testSIMD (cf. sa définition, page 202) qui crée un vecteur paqueté du masque que l'on utilise aussitôt dans la fonction de déplacement conditionnel cndmoveSIMD (cf. sa définition, page 203). Cette construction est une technique courante pour exprimer l'expression conditionnelle **if then else** pour les données SIMD.

La fonction map2 applique la fonction λ sur les éléments de deux streams. Les fonctions mkAr2DPVec et mkAr2DFromAr2DPVec sont utilisées pour vectoriser et dé-vectoriser les arrays, respectivement, et les fonction **elems** et **listArray** sont les fonctions standards du Haskell pour le passage d'un array à un stream et vice versa, respectivement.

7.4 Approche utilisant les macro blocs avec la transposition directe

Le skeleton algorithmique de propagation propAlgSQR4 que nous avons utilisé jusqu'à présent n'employait la transposition de données qu'après avoir accompli les phases entières de la propagation. Pour pouvoir plus profiter de la localité des données lors du parcours de l'image, nous pouvons envisager la construction des algorithmes qui travailleraient à l'échelle des macro blocs dont la taille serait égale à la taille du registre multimédia de notre architecture et où nous effectuerions deux phases de propagation et une transposition par diagonale en même temps sans avoir besoin d'écrire les données dans la mémoire principale.

Le principe de cette technique est illustré par la fig. 7.10 qui présente la chaîne de traitement avec les données représentées en tant que flux et les opérations qui sont effectuées sur ces données en tant que kernels d'exécution. Nous n'allons pas donner de description formelle à cette technique à l'échelle de l'image toute entière. Nous présentons seulement le principe de fonctionnement de cette approche à l'échelle de macro bloc car nous pensons que sa description formelle peut être déduite à partir du style de travail que nous avons présenté pour le skeleton propAlgSQR4.

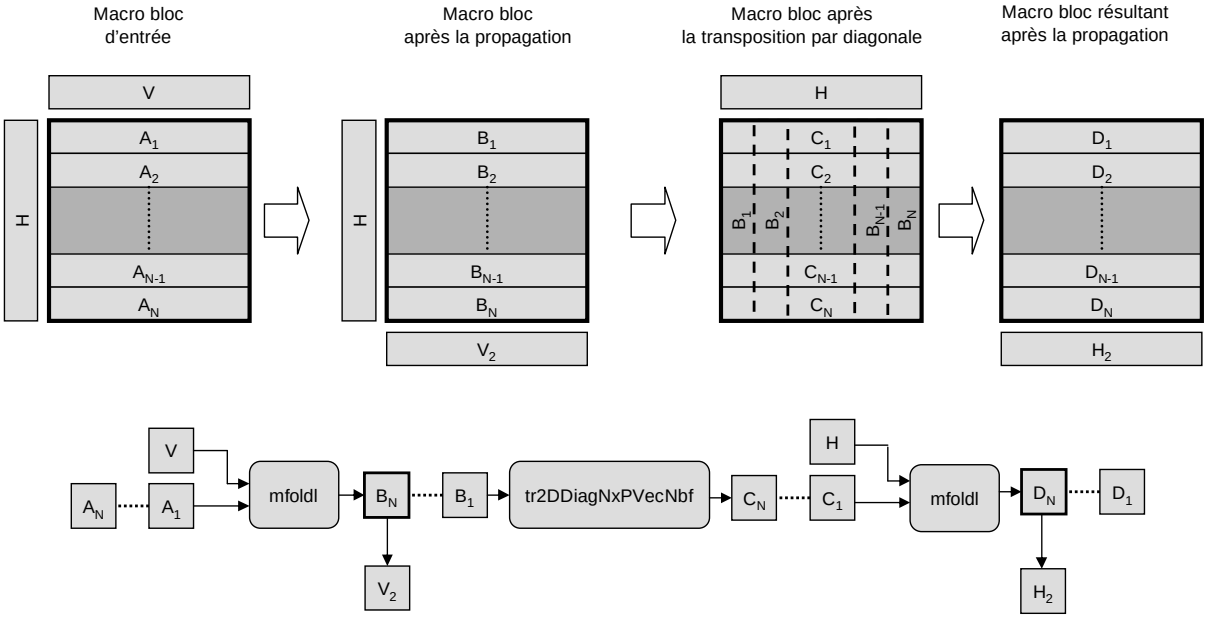


FIG. 7.10 : La chaîne de traitement d'un macro bloc où la première propagation est suivie directement par la transposition par diagonale et par la deuxième propagation

Tout d'abord, nous appliquons la première phase de propagation (verticale), exprimée par la fonction mfoldl. Cette fonction utilise l'élément V comme entrée pour la propagation. Cet élément correspond

au résultat de la propagation verticale précédente et nous l'utilisons en tant que valeur initiale pour la propagation (au bord de l'image, nous utiliserons le schéma de propagation décrit par la fonction mfoldl1 qui ne travaille pas avec un tel élément).

Après avoir terminé la propagation, nous gardons la valeur du dernier élément B_N du stream résultant dans une mémoire temporaire V_2 pour pouvoir la réutiliser lors de la prochaine propagation dans la même direction.

Sur le stream B_i, nous appliquons la transposition par diagonale, exprimée par le kernel du calcul tr2DDiagNxPVecNbf (cf. l'algorithme 6.4) qui utilise, rappelons-le, les fonctions shuffle pour exécuter la transposition en $N\log_2 N$ opérations. Le stream résultat C_i de cette transposition est mis en relation avec la valeur résultante H de la propagation horizontale[1] précédente pour ensuite appliquer une deuxième phase de propagation verticale des valeurs à l'aide de la fonction mfoldl (les mêmes suppositions se mettent en place pour le travail sur les bords de l'image).

Les résultats de cette propagation C_i sont les résultat semi-finaux. Nous prenons la valeur du dernier élément D_N et nous la sauvegardons comme H_2 pour l'évaluation suivante du macro bloc voisin. Les valeurs de ce stream résultant sont écrites dans la mémoire. Ainsi, nous avons obtenu le résultat de deux phases de propagation au niveau du macro bloc.

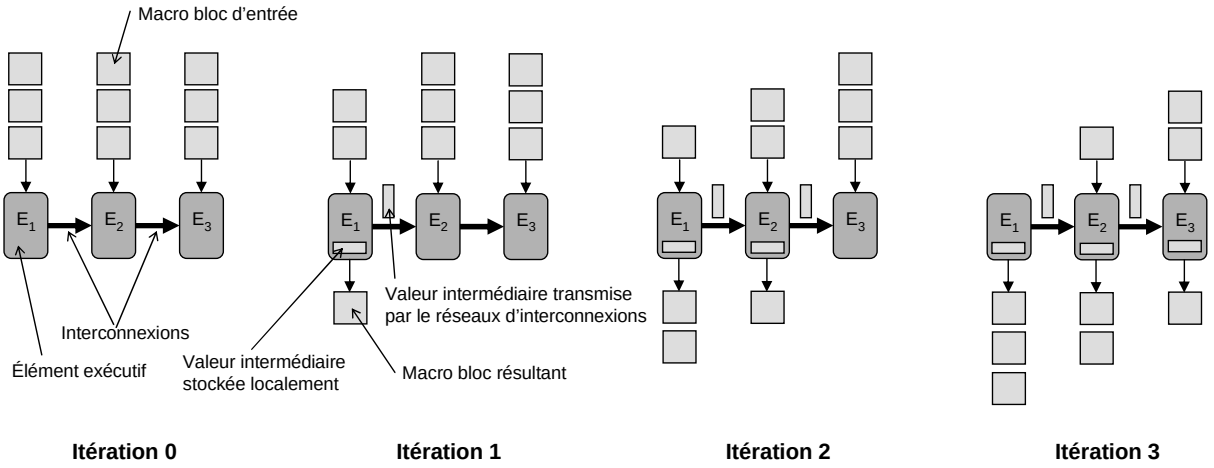


FIG. 7.11 : Propagation à l'échelle des macro blocs lors du calcul avec plusieurs unités exécutives. Une des valeurs intermédiaires est stockée localement dans l'unité, la deuxième est transmise par le réseau d'interconnexion à l'unité suivante.

Il faut noter que l'implémentation physique de cette approche demande les connaissance de l'architecture cible car les valeurs intermédiaires de la propagation doivent être gérées différemment sur les architectures à un seul fil d'exécution où nous avons besoin d'une mémoire temporaire pour les sauvegarder mais qu'elles peuvent être passées à l'unité d'exécution suivante (e.g. sur les architectures à plusieurs fils d'exécution), comme l'illustre la fig. 7.11. Si nous n'avons pas d'architecture capable de travailler dans une telle configuration, l'utilisation de la mémoire temporaire pour stocker les résultats intermédiaires de la propagation doit être envisagée.

7.5 Notes sur l'implémentation, résultats expérimentaux

Les implémentations de la fonction distance et des nivellements que nous avons conçues ont été initialement destinées au processeur SuperH SH-5 qui bénéficie de 64 registres de 64 bits. Ces implémentations ont été incluses dans l'outil MorphoMedia que nous avons développé afin d'obtenir l'indépendance

[1] Après avoir effectué la transposition par diagonale, la propagation est effectuée, à l'échelle du macro bloc, verticalement mais elle correspond à une propagation horizontale dans l'image

sur une architecture donnée. C'est pourquoi nous avons pu les compiler facilement pour l'architecture Intel IA-32, qui fait le point de référence dans cette thèse, en nous appuyant sur la technologie Intel MMX (64 bits) et SSE2 (128 bits), le dernier implémentant les fonctions SIMD pour les maxima et les minima utilisés dans la morphologie mathématique.

Tout d'abord, nous présentons les résultats de l'implémentation de la fonction distance sur la grille carrée et ayant 4-voisins par pixel, q.v. tab. 7.1. Il s'agit d'un algorithme de base qui est bien connu. Nous voulions obtenir les temps d'exécution chiffrés pour cette fonction distance pour pouvoir les comparer par la suite avec les temps obtenus pour les nivellements. En consultant la table 7.1 nous constatons que pour une image de 99 ko, nous obtenons sur un processeur Intel Pentium 4 à 2.4 GHz un temps d'exécution de 0.34 ms. Ce temps est le plus favorable possible car l'algorithme présenté travaille avec l'image de sortie ayant les éléments de 8 bits ; ce qui peut, pour certains types d'images, poser des problèmes d'insuffisance du type de stockage, mais nous le présentons ici pour une autre raison – les algorithmes des nivellements présentés par la suite travaillent avec les images d'entrée et de sortie de 8 bits et cette implémentation de la fonction distance peut nous servir en tant que référence pour la comparaison.

Fonction distance, grille carrée, 4-voisins		
méthode d'implémentation	temps en ms	taux d'accélération
générique	2.42	1.0
par macro blocs utilisant la transposition par diagonale directe	0.34	7.1

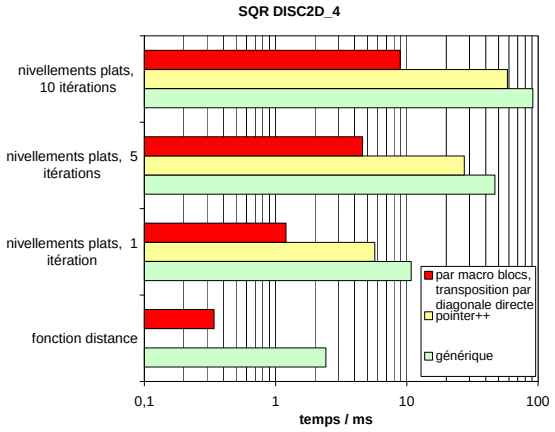
Image $352 \times 288 \times 8\ bits$ = 99 ko, l'image d'entrée et l'image de sortie sont de 8 bits. L'algorithme *générique* utilise les fonctions getpixel()/setpixel() et la propagation en sens vidéo/anti-vidéo ; l'algorithme *par macro blocs* est optimisé pour les types multimédia de 64 bits et utilise la transposition directe à l'échelle des macro blocs. Exécuté 1000 fois en trois réalisations, le temps présenté est le moyen de la meilleure réalisation. Processor Intel Pentium 4 à 2.4 GHz, mémoire cache L2 = 512 ko ; système d'exploitation Linux Mandrake 9.2 ; compilateur Intel ICC 8.1 pour Linux.

TAB. 7.1 : Résultats expérimentaux pour diverses implémentations de la fonction distance sur la grille carrée et 4-voisins par pixel

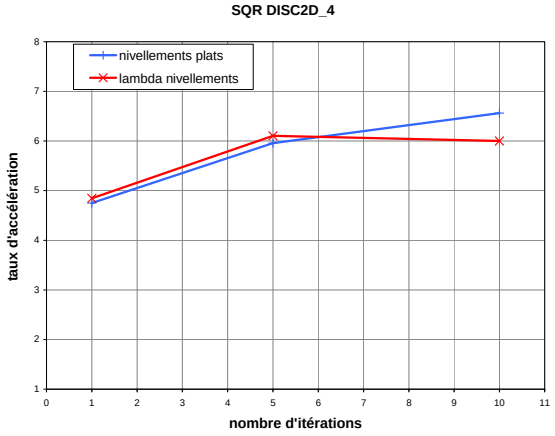
Les résultats des diverses implémentations des nivellements plats et des lambda-nivellements avec la valeur $\lambda = 1$ sont présentés dans la table 7.2. Les conditions des tests (dimensions d'image, processeur etc.) pour les nivellements sont identiques à celles pour la fonction distance. En consultant les temps de l'implémentation la plus rapide *par macro blocs* pour 1 itération – 1.2 ms pour les nivellements plats et 1.3 ms pour les lambda-nivellements – et en les comparant avec la valeur obtenue pour la fonction distance (0.34 ms), nous pouvons constater que l'exécution de l'algorithme des nivellements est à peu près 3.5 à 4 fois plus longue que celle de la fonction distance correspondante.

En ce qui concerne les différences entre les diverses implémentations des nivellements, nous avons choisi comme référence l'implémentation *pointeur++* (son taux d'accélération est égal à 1.0 est présenté en gras). Le masque, qui est une des images d'entrée de l'algorithme des nivellements, n'est pas modifié par ce dernier. Ainsi, sa transposition peut être effectuée en avance et seulement une fois pour toutes les itérations. Les temps que nous présentons pour l'implémentation *par macro blocs*, reflètent ce fait et nous mentionnons entre parenthèses les temps incluant cette transposition préalable. La fig. 7.12(a) présente une comparaison graphique des temps d'exécution pour différentes implémentations de la fonction distance et des nivellements plats.

Sachant que le parallélisme SIMD que nous utilisons est de 8 éléments de 8 bits (implémentation utilisant les types multimédia de 64 bits) et en prenant en compte que nous effectuons deux transpositions par diagonale dans notre implémentation *par macro blocs* qui ne sont pas présentes dans l'implémentation *pointer++*, le taux d'accélération des nivellements est de 4.8 pour 1 itération et il augmente avec



(a) Comparaison des temps d'exécutions de la fonction distance et des nivellements plats



(b) Taux d'accélération pour les nivellements

FIG. 7.12 : Résultats expérimentaux des algorithmes dépendant du sens prédéfini du parcours de l'image

Nivellements, grille carrée, 4-voisins							
type	implémentation	1 itération temps ms	1 itération taux d'accélération	5 itérations temps ms	5 itérations taux d'accélération	10 itérations temps ms	10 itérations taux d'accélération
plats ($\lambda = 0$)	générique	10.8	—	46.9	—	91.3	—
	pointeur++	5.7	**1.0**	27.4	**1.0**	58.4	**1.0**
	par MB	1.2 (1.8)	4.8 (3.2)	4.6 (5.2)	6.0 (5.3)	8.9 (9.5)	6.6 (6.1)
lambda ($\lambda = 1$)	générique	11.8	—	51.9	—	102.3	—
	pointeur++	6.3	**1.0**	29.3	**1.0**	55.2	**1.0**
	par MB	1.3 (1.9)	4.8 (3.3)	4.8 (5.4)	6.1 (5.4)	9.2 (9.9)	6.0 (5.6)

Légende : MB = macro blocs, TD = transposition par diagonale ; Image $352 \times 288 \times 8\ bits$ = 99 ko, l'image d'entrée et l'image de sortie sont de 8 bits. L'algorithme *générique* utilise les fonctions getpixel()/setpixel() et la propagation en sens vidéo/anti-vidéo ; l'algorithme *pointeur++* est une analogie de l'algorithme *générique* mais il utilise explicitement les pointeurs ; l'algorithme *par MB* est optimisé pour les types multimédia de 64 bits et utilise la transposition directe à l'échelle des macro blocs ; entre parenthèses nous présentons les temps et les taux d'accélération qui incluent la transposition préalable de l'image du masque. Exécuté 1000 fois en trois réalisations, le temps présenté est le moyen de la meilleure réalisation. Processeur Intel Pentium 4 à 2.4 GHz, mémoire cache L2 = 512 ko ; système d'exploitation Linux Mandrake 9.2 ; compilateur Intel ICC 8.1 pour Linux.

TAB. 7.2 : Résultats expérimentaux pour diverses implémentations des nivellements sur la grille carrée et 4-voisins par pixel

d'avantage d'itérations – nous obtenons le taux d'accélération de 6.6 pour 10 itérations. La fig. 7.12(b) présente graphiquement les taux d'accélérations des nivellements plats et lambda pour l'implémentation *par macro blocs* qui utilise la transposition directe ; l'algorithme *pointeur++* fait la référence (taux d'accélération égal à 1).

Remarquons l'écart important entre l'implémentation *générique* (qui parcourt l'image en sens vidéo et anti-vidéo, utilise les fonctions d'accès aux pixels (setpixel(), getpixel()) et peut travailler avec n'importe quelle grille) et l'implémentation *pointeur++* (qui parcourt également l'image en sens vidéo et anti-vidéo mais qui est spécialisée pour la grille carrée de 4-voisins).

Nous présentons une des applications possibles des nivellements pour le filtrage du flux vidéo lors d'une vidéo conférence, q.v. la fig. 7.13(a) présentant l'original et la fig. 7.13(b) présentant les résultats.

Nous travaillons avec la luminance (cf. la fig. 7.13(e)) et pour obtenir le marqueur pour les nivellements, le filtre alterné séquentiel y est appliqué, cf. la fig. 7.13(f). L'algorithme des nivellements est ensuite appliqué sur l'image entière, q.v. 7.13(g). À la fin, nous appliquons une simple méthode du masquage conditionnel (cf. l'image du masque sur la fig. 7.13(h)) qui, dans l'image résultant de ce filtrage, laisse l'intérieur de la zone d'intérêt intact et remplace la zone extérieure par les pixels filtrés.

(a) Original

(b) Résultat

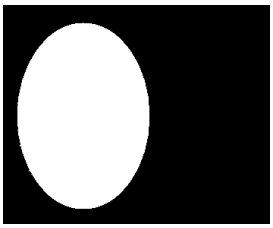
(c) Détail de l'original

(d) Détail du résultat

(e) Luminance de l'original

(f) Marqueur - résultat du filtre alterné de taille 7

(g) Nivellements



(h) Masque

FIG. 7.13 : Application des nivellements au filtrage des images conditionné par un masque, dimensions de l'images 382 × 288, nivellements effectués sur la luminance.

7.6 Récapitulation

Les méthodes que nous avons présentées dans ce chapitre sont les méthodes qui exploitent implicitement le parallélisme des données lors de la propagation régulière des valeurs dans l'image. Le but principal qui nous a conduit à présenter ces méthodes était de démontrer que si nous voulons transformer les méthodes utilisant une telle propagation et travaillant avec les éléments de base à des méthodes utilisant les données paquetées, la démarche n'est pas triviale.

Les problèmes que nous rencontrons lors de l'utilisation des types paquetés pour les traitements sont dûs à l'incapacité d'extraire de la mémoire les données paquetées dans le sens perpendiculaire à celui d'adressage. C'est pourquoi nous avons démontré sur l'exemple de la propagation sur la grille carrée et le voisinage défini par le point central et 4 voisins la manière dont nous pouvons effectuer la propagation dans les 4 directions – en faisant appel à une méthode de changement de stockage et à

seulement deux types de propagation différents. Ainsi, nous avons pu réutiliser les idées exposées dans le chapitre 6, qui était consacré aux permutations SIMD des données, et dont nous avons repris l'algorithme tr2DDiagNxPVecNbf de la transposition rapide d'un macro bloc à l'aide des instructions multimédia.

Pour pouvoir modéliser les propagations formellement, nous avons introduit les skeletons algorithmiques de base pour la propagation des valeurs, mfoldl et mfoldl1. Ils nous ont servi lors de la construction du modèle formel de la propagation à l'intérieur d'un macro bloc composé des vecteurs paquetés et défini par la fonction pGenMB, cf. page 152, et à partir duquel nous avons pu construire un algorithme général de la propagation SIMD en 4-voisinage sur la grille carrée – propAlgSQR4, cf. page 153.

L'emploi plus élaboré de la propagation à l'échelle des macro blocs qui utilise la transposition des données directement après avoir propagé les valeurs dans un macro bloc permet d'enchaîner tout de suite la deuxième propagation en économisant ainsi le nombre des transferts de données entre l'unité centrale et la mémoire. Nous avons également présenté comment cette approche pourrait être adoptée pour les architectures parallèles où l'exécution se poursuivrait sur plusieurs unités exécutives en même temps.

Nous avons présenté deux groupes d'algorithmes dont le fonctionnement est dépendant du sens du parcours et qui utilisent la propagation régulière des valeurs dans l'image. Il s'agit de la fonction distance et des nivellements. Dans la section qui exposait les résultats expérimentaux des implémentations que nous avons effectuées, nous avons pu constater un gain de performance qui n'est pas négligeable mais prévisible vu le nombre d'éléments (8) que nous pouvions traiter par les instructions SIMD en même temps.

Le sujet que nous avons exposé dans ce chapitre et les résultats que nous avons obtenus peuvent être appréciés dans les applications qui utilisent une architecture multimédia, qui sont contraintes par le temps d'exécution, et dont nous visons prioritairement les applications pour le temps réel.

Les idées qui sont connexes à ce chapitre et qui restent, pour l'instant, inexplorées sont principalement celles qui touchent les architectures parallèles : il serait à envisager d'explorer les stratégies de construction des algorithmes pour les architectures qui peuvent assurer l'exécution concurrente des tâches, notamment les architectures parallèles à plusieurs fils d'exécution. D'autres sujets à explorer sont relatifs à la construction des architectures dédiées qui implémenteraient la structure des algorithmes présentés directement dans le matériel (e.g. FPGA). On pourrait alors concevoir les blocs opérationnels exploitant mieux la localité de données, définir un schéma d'interconnexion fixes qui implémenterait le changement de stockage de données et envisager l'utilisation du parallélisme SIMD plus important (> 8).

Du point de vue des algorithmes, la façon de travailler présentée pour la fonction distance et pour les nivellements peut être employée dans tous les algorithmes morphologiques faisant appel à la *reconstruction* et à tous les algorithmes dérivés de cette dernière (e.g. l'algorithme de bouchage des trous, l'algorithme d'extraction des maxima ou des minima de l'image etc.).

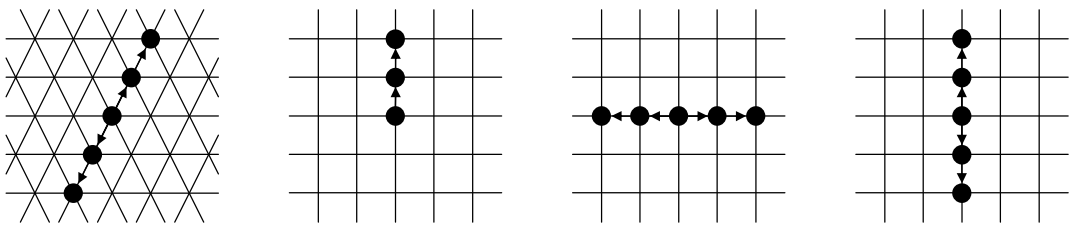
Cette page est blanche par intention

CHAPITRE 8

Algorithmes de la dilatation/érosion pour les éléments structurants de la forme d'un segment

Ce chapitre est consacré aux approches des calculs des opérations morphologiques qui combinent toutes les idées que nous avons présentées jusqu'à présent dans les chapitres précédents. D'un côté, l'exécution de ces algorithmes ne dépend pas, à l'échelle des macro blocs, de l'ordre particulier de leur traitement ; ainsi, nous pouvons nous appuyer sur les idées décrite dans le chapitre 5, page 99, consacré aux algorithmes morphologiques non-dépendants du sens de parcours. D'un autre côté, une partie d'exécution de ces algorithmes, celle qui concerne le traitement à l'intérieur d'un macro bloc, est dépendante du sens de parcours. Ce sens de parcours est connu en avance et est défini par l'élément structurant, exactement comme c'était le cas pour les algorithmes décrits dans le chapitre 7, page 147, traitant ce sujet. De plus, l'exécution de certains algorithmes que nous allons décrire va s'appuyer sur les techniques de changement de l'axe de stockage des données afin de pouvoir accéder aux données vectorielles dans l'axe perpendiculaire à l'axe de stockage de ces données dans la mémoire ; par conséquent, nous allons réutiliser également les idées qui ont été décrites dans le chapitre 6, page 127, consacré à ces techniques particulières.

Lors du travail avec les images 2D (ou plus), nous comprenons sous le terme des *éléments structurants linéaires* les éléments structurants constitués des vecteurs de déplacement parallèles à une droite. Parmi eux, une place privilégiée est détenue par des *segments*. Un segment est un élément structurant qui désigne un groupe de pixels qui sont connectés ; pour une grille et un voisinage donnés, il doit exister un chemin entre un pixel de ce groupe et tous les autres pixels de ce groupe ; cela doit être valable pour tous les pixels appartenant au groupe. Ce sont les segments qui seront notre centre d'intérêt dans ce chapitre et auxquels sont destinés les algorithmes décrits par la suite. La figure 8.1 illustre, sur quelques exemples, les éléments structurants de la forme d'un segment.



(a) Segment symétrique, grille hexagonale — *(b)* Segment directionnel, grille carrée — *(c)* Segment symétrique horizontal, grille carrée — *(d)* Segment symétrique vertical, grille carrée

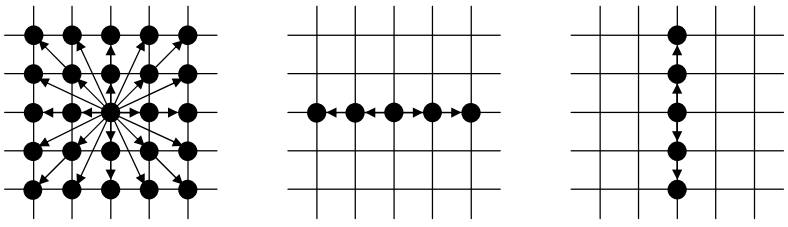
FIG. 8.1 : Exemples des éléments structurants ayant la forme d'un segment

Pour les besoins de cette thèse où nous expliquons le principe de la construction des algorithmes pour les dilatations et les érosions par un segment pour les architectures parallèles avec les fonctionnalités SIMD, nous allons nous restreindre aux segments définis sur la grille carrée, qui suivent les directions des axes de l'image et qui sont symétriques, cf. la fig. 8.1(c) et la fig. 8.1(d). La gestion des cas plus généraux, tels que les segments directionnels (cf. la fig. 8.1(b)) ou les segments inclinés est également envisageable mais nous ne nous consacrons pas à ces algorithmes car nous pensons qu'ils sont déductibles à partir de la description que nous nous apprêtons à donner par la suite.
Les éléments structurants de la forme d'un segment sont au centre de notre intérêt pour deux raisons :

- La première est relative aux architectures matérielles qui sont destinées à l'exécution de nos opérations morphologiques. En effet, les algorithmes pour le calcul rapide des dilatations / érosions par un segment sont faciles à adapter pour les architectures multimédia avec les capacités SIMD et en les utilisant, nous pouvons obtenir un gain de performance important sur les architectures existantes grand public.
- La deuxième raison est relative à la théorie de la morphologie mathématique et au fait qu'en utilisant les propriétés des opérateurs morphologiques, nous pouvons décomposer l'opération de dilatation / d'érosion employant un élément structurant complexe de grande taille en une séquence des dilatations / érosions employant des éléments structurants unidimensionnels comptant un nombre d'éléments beaucoup moins élevé. Pour l'exemple de l'élément structurant B qui est illustré sur la fig. 8.2(a), nous pouvons procéder à une décomposition de l'opération de dilatation en une séquence de dilatations par les éléments structurants B_H (q.v. la fig. 8.2(b)) et B_V (q.v. la fig. 8.2(c)) ; en ce qui concerne l'élément structurant original, nous parlons de sa *décomposition*. L'opération dilatation peut ainsi être décrite comme :

$$\delta_B = \delta_{B_H} \circ \delta_{B_V} = \delta_{B_V} \circ \delta_{B_H} \tag{8.1}$$

  Cette technique est un outil de base pour l'implémentation de toutes les opérations qui utilisent abondamment les dilatations, les érosions, les ouvertures et les fermetures, notamment les implémentations des filtres alternés séquentiels et toutes les techniques des ouvertures pour les mesures de la granulométrie. Son apport exprimé par la réduction de nombre des opérations qui doivent être effectuées et leur impact sur la complexité est évident ; nous pouvons l'illustrer également sur l'exemple de la fig. 8.2 qui présente un élément structurant original de 25 vecteurs de déplacement (cf. fig. 8.2(a)) décomposé en deux éléments structurants, chacun de 5 vecteurs de déplacement, 10 en total (cf. fig. 8.2(b).



(a) Élément structurant B à décomposer | *(b)* Élément structurant horizontal B_H | *(c)* Élément structurant vertical B_V

FIG. 8.2 : Décomposition d'un élément structurant sur la grille carrée

8.1 Approche itérative

Nous commencerons notre explication en mentionnant l'approche itérative à l'implémentation des opérations de dilatation / érosion car c'est cette approche qui fait référence aux autres algorithmes. Elle

suit directement la définition mathématique pour la construction des éléments structurants de taille $n, n > 1$ à partir des éléments structurants de taille unité (1) en appliquant l'élément structurant sur lui-même en $n - 1$ itérations (exprimé par le symbole $\bigcirc$) et en exploitant ainsi l'homothétie de ces éléments structurants :

$$\delta_{B_n} = \bigcirc_{n-1} \delta_{B_1} \tag{8.2}$$

Il s'agit de la plus simple approche à l'implémentation du point de vue du temps qui doit être consacré au développement d'un tel algorithme car il nous suffit de réutiliser les algorithmes implémentant la dilatation/érosion que nous avons mentionnés dans le chapitre 5 consacré aux algorithmes morphologiques non-dépendants du sens du parcours. Formellement, notre algorithme en Haskell va refléter ce style de travail. L'algorithme 8.1 de la dilatation par un segment de la taille *n* sur la grille carrée (défini par la fonction dilSQRHomothetic) qui se base sur l'élément structurant unité décrit par la liste des vecteurs de déplacement *ngb* et qui prend en compte la valeur constante *brdval* pour la gestion des bords de l'image peut être construit par $(n - 1)$ applications successives de l'algorithme l'algorithme dilSQR (décrit dans la section 5.1.1.2, page 102). Pour plus d'informations sur l'exécution itérative en Haskell et pour l'explication des raisons de l'utilisation des fonctions last, take et iterate dans l'algorithme dilSQRHomothetic, cf. Annexe A, page 199.

Algorithme 8.1 : dilSQRHomothetic, Algorithme dilSQRHomothetic de la dilatation pour la grille carrée par un segment en utilisant l'approche itérative

```
1 dilSQRHomothetic :: (Ord α) ⇒ I → α → Ngb → Ar (I,I) α → Ar (I,I) α
2 dilSQRHomothetic n brdval ngb ar = last ∘ (take n) ∘ (iterate (dilSQR brdval ngb)) $ ar
```

8.2 Approche employant les algorithmes à réutilisation des valeurs

La réutilisation des valeurs intermédiaires calculées lors de l'évaluation des opérations morphologiques pour les éléments structurants ayant la forme d'un segment est le sujet de nombreux articles et travaux scientifiques. L'existence de ces publications prouve l'importance de la place qu'elles occupent dans la morphologie mathématique et de leur utilité apportée aux implémentations sur une architecture matérielle donnée.

Parmi les méthodes présentées, nous retrouvons l'algorithme[vH92], présenté en 1992 par Marcel van Herk, qui assure l'implémentation des dilatations et des érosions par un segment de taille quelconque et dont la complexité ne dépend pas de la taille de ce segment. La même idée pour le calcul des opérations morphologiques est évoquée dans un autre article[GW93] dont les auteurs sont Joseph Gil et Michael Werman et qui, bien que publié plus tard (1993), a été déposé plus tôt pour la publication que celui de van Herk. Ce fait peut faire croire que les deux travaux ont été créés indépendamment et c'est pourquoi on désigne cet algorithme comme "l'*algorithme de van Herk-Gil-Werman*[1]". Dans l'utilisation courante, on le désigne souvent comme l'algorithme de (seulement) van Herk, effaçant ainsi à tort la mention et les apports des deux autres auteurs. Pourtant, la description dans l'article original de Gil et Werman est plus théorique et a vraiment la notion d'une idée tandis que l'article original de van Herk décrit l'algorithme plus pratiquement et nous le jugeons plus compréhensible.

Le fonctionnement de cet algorithme a été décrit[Soi03] et discuté dans plusieurs travaux traitant des améliorations[GAA97] et les das d'utilisation mais également dans les articles[GK02] qui ont réutilisé son idée pour les ouvertures et les fermetures ou les articles[NHS97, NH00, SBJ96] qui étudient son idée pour le travail avec des segments inclinés ayant une orientation arbitraire en exploitant l'algorithme[Bre65] de Bresenham de la création des lignes dans le domaine digital.

[1] Cette appellation est à trouver dans l'article[GK02] de Gil et Kimmel, le premier étant l'auteur de l'algorithme.

Les travaux qui sont liés au nom de Marc Van Droogenbroeck proposent des méthodes améliorés pour le calcul rapide des érosions / dilatations et les ouvertures et les fermetures par des segments. Nous citons l'article que cet auteur a publié en collaboration avec Hugues Talbot[DT96] et un autre article publié en collaboration avec M. Buckley[DB05]. De plus, ces articles proposent les résultats des études comparatives avec d'autres algorithmes existants du calcul des opérations morphologiques pour les segments. Les implémentations de leurs algorithmes sont librement disponibles dans la bibliothèque des algorithmes "The libmorpho library"[DD06] qui mentionne dans sa documentation également d'autres résultats des tests comparatifs.

Nous avons choisi l'algorithme de van Herk-Gil-Werman pour démontrer les principes de la construction de l'implémentation de ce type d'algorithmes pour les architectures multimédia avec les capacités SIMD.

8.2.1 Principe de l'algorithme de van Herk-Gil-Werman

Le fonctionnement de l'algorithme de van Herk-Gil-Werman (mentionné par l'abréviation *HGW* dans la suite) était décrit originalement[vH92] pour les éléments structurants qui ont la forme d'un segment symétrique et qui, par conséquent, désignent le nombre impair des pixels voisins. L'algorithme HGW est composé de 3 phases. Les deux premières préparent les valeurs intermédiaires des maxima (pour la dilatation) / minima (pour l'érosion) par la propagation des valeurs ; cette propagation est effectuée à l'échelle des blocs des pixels et non de l'image entière. Dans la troisième phase, nous évaluons les résultats finaux à partir des résultats intermédiaires. La figure 8.3 illustre une vue globale sur ce fonctionnement en désignant par les flèches les pixels de source et de destination pour l'opération max / min et en montrant ainsi la dépendance entre les résultats intermédiaires lors de l'évaluation.

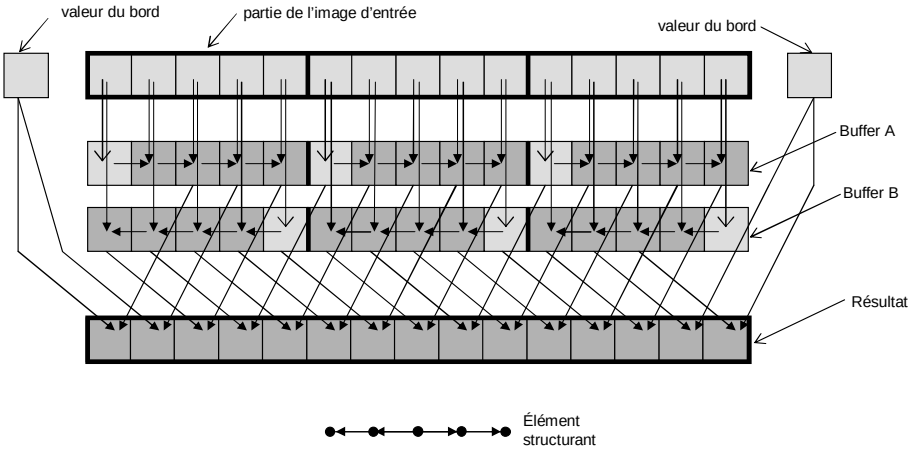


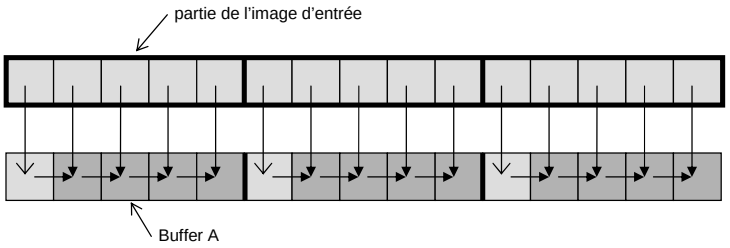
FIG. 8.3 : Schéma de fonctionnement de l'algorithme de van Herk-Gil-Werman

Expliquons le fonctionnement de cet algorithme plus en détail. Tout d'abord, nous procédons, dans la direction parallèle à l'orientation de l'élément structurant, au découpage de l'image d'entrée en blocs de pixels qui sont de la même taille que celle de l'élément structurant et qui ont la dimension 1 dans la direction perpendiculaire. C'est à l'échelle de ces macro blocs que le traitement est indépendant du sens de parcours et que c'est à l'intérieur de ces macro blocs que nous allons effectuer la propagation des valeurs. La figure 8.3 illustre ce découpage sur une partie de l'image, qui correspond à une ligne de cette image, pour l'élément structurant horizontal. Les blocs découpés de 5 pixels sont délimités par la bordure épaisse.

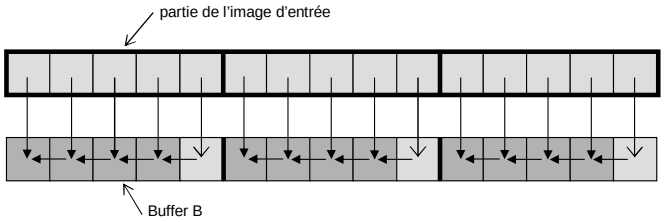
Lors de l'explication du fonctionnement, nous allons supposer que l'image a des dimensions des multiples de la taille de l'élément structurant. On obtiendra ainsi le découpage parfait de l'image aux macro blocs. Les cas spéciaux de l'image qui ne pourraient pas être découpés parfaitement et qui contiendraient

à chaque ligne un macro bloc de taille plus petite et connexe au bord de l'image seraient gérés par le kernel du calcul différent, spécifique à la zone de l'image connexe au bord. Ainsi, nous suivons l'idée de la décomposition du traitement en traitements de la zone de l'intérieur et en traitement de la zone du bord, comme présentés dans le chapitre 5, page 102 dans la section 5.1.2 qui était dédié à cette problématique.

La première phase, p_1, de cet algorithme qui nous fournit la première série des résultats intermédiaires, utilise la propagation à l'intérieur des macro blocs dans une direction, choisie par convention comme la direction suivant le sens des index montants. La figure 8.4(a) illustre cette situation. La propagation des valeurs est effectuée de la même manière que précédemment mentionnée dans le chapitre 7, page 7.2, dans la section 7.2 dédié aux skeletons applico-rédictifs mfoldl et mfoldl1. Donc, il est naturel d'utiliser ces skeletons pour exprimer cette phase de l'algorithme HGW dans le formalisme fonctionnel. Les résultats intermédiaires que nous obtenons après l'application de cette phase sont groupés, sur la fig. 8.4(a) en tant que *Buffer A*.



(a) Première phase, p_1



(b) Deuxième phase, p_2

FIG. 8.4 : Phases p_1 et p_2 de propagation des valeurs de l'algorithme de van Herk-Gil-Werman

Avant de donner la description formelle pour la première phase, nous présentons informellement le fonctionnement de la deuxième phase, p_2, de l'algorithme HGW. Elle est, en effet, analogue à la première et elle utilise la même manière de propagation des valeurs à l'exception du sens de la propagation qui est opposé. La figure 8.4(b) illustre cette situation.

Nous pouvons explorer les points communs est construire ainsi la description formelle d'une phase généralisée de la propagation des valeurs de l'algorithme HGW. La fonction phaseHGW est dédiée à ce but et définit cette phase généralisée :

```
phaseHGW  ::  (Ord α) ⇒ [Char] → (α → α → α) → Ar (I,I) α → Ar (I,I) α
phaseHGW how f mb =   (array (bounds $ mb))
                    ∘ (zip ixs)
                    ∘ (mfoldl1 f)
                    ∘ (map ( mb! ))
                    $ ixs
  where  ixs = streamAr2D how mb
```

Le choix exact de la propagation est spécifié par le paramètre *how*. En regardant bien cette fonction, nous nous apercevons que son corps est, en effet, identique à celui de la fonction pGenMB (cf. sa définition dans le chapitre 7, page 152) qui décrivait la propagation SIMD dans un macro bloc composé des vecteurs paquetés. Les deux fonctions diffèrent dans leurs signatures de types et, par conséquent, dans l'usage

prévu. Tant que la fonction pGenMB a été conçue directement pour le travail avec les vecteurs paquetés, la fonction d'une phase généralisée phaseHGW est destinée tout d'abord au travail avec les éléments de base. C'est en effet cette similitude qui va lier les algorithmes de la propagation SIMD et les stratégies de parallélisation de l'algorithme de van Herk-Gil-Werman.

Pour pouvoir dériver les phases concrètes p_1 et p_2 de la propagation à partir de la phase généralisée, nous nous baserons sur le paramètre *dir* qui exprimera textuellement la direction du segment, *"Fst"* / *"Snd"* si l'élément structurant suit la direction de la première / deuxième coordonnée, respectivement. La première phase de l'algorithme HGW est définie par la fonction phase1 :

```
phase1  ::  (Ord α) ⇒ [Char] → (α → α → α) → Ar (I,I) α → Ar (I,I) α
phase1 dir f mb = phaseHGW ("FW"++dir) f mb
```

et la deuxième phase de l'algorithme HGW est définie par la fonction phase2 :

```
phase2  ::  (Ord α) ⇒ [Char] → (α → α → α) → Ar (I,I) α → Ar (I,I) α
phase2 dir f mb = phaseHGW ("BW"++dir) f mb
```

Les deux fonctions utilisent à l'interne la fonction généralisée phaseHGW et la spécifient pour le parcours *en avant* (par "FW") dans le cas de la phase p_1 et pour le parcours *en arrière* (par "BW") dans le cas de la phase p_2. Notons que les deux premières phases peuvent être exécutées en même temps en exploitant ainsi le parallélisme des tâches.

La troisième phase utilise les résultats intermédiaires obtenus par l'application de la phase p_1 et p_2 et les combine dans le résultat final de l'algorithme. La fig. 8.5 illustre son fonctionnement.

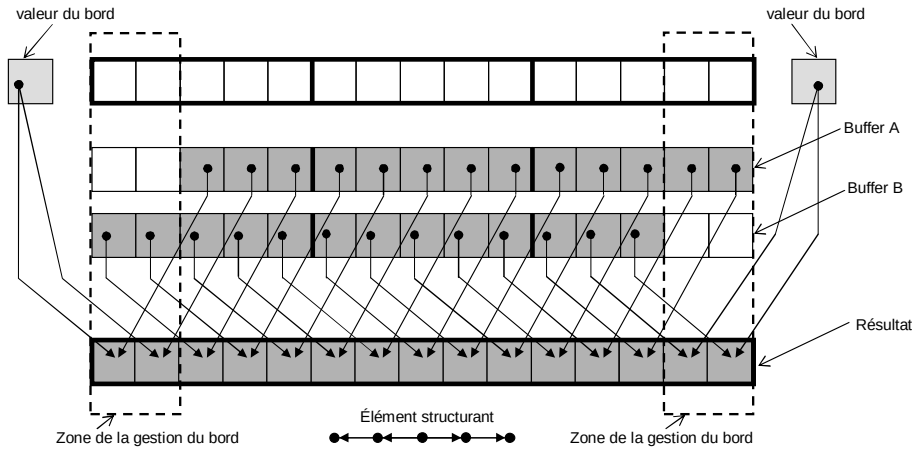


FIG. 8.5 : Schéma de fonctionnement de l'algorithme de van Herk, phase p_3

Elle ne travaille pas à l'échelle des macro blocs mais à celle d'une ligne (colonne). Étant donné k le nombre des pixels désignés par l'élément structurant, nous extrayons, pour une position donnée x d'une dimension correspondant au pixel central à l'échelle d'une ligne (colonne), les pixels avec l'index $x + \frac{(k-1)}{2}$ à partir du buffer A et les pixels avec l'index $x - \frac{(k-1)}{2}$ à partir du buffer B. Puisqu'il s'agit du calcul utilisant les vecteurs de déplacement pour évaluer la valeur d'un pixel, tous les phénomènes de traitement non-dépendants du sens du parcours par les éléments structurants, comme présentés dans le chapitre 5, page 99, entrent en jeu, notamment ceux qui sont connexes à la gestion des pixels aux bords de l'image. La figure 8.6 illustre ce travail sur l'exemple de trois pixels, un placé dans la zone intérieure de l'image et d'autres placés dans la zone de traitement des bords de l'image.

Nous présentons la définition de la fonction phase3Gen qui effectue cette troisième phase et qui, pour extraire une valeur à partir des buffers temporaires buf_A et buf_B, utilise l'approche naïve au traitement des effets de bord, représentée par la fonction sampGen :

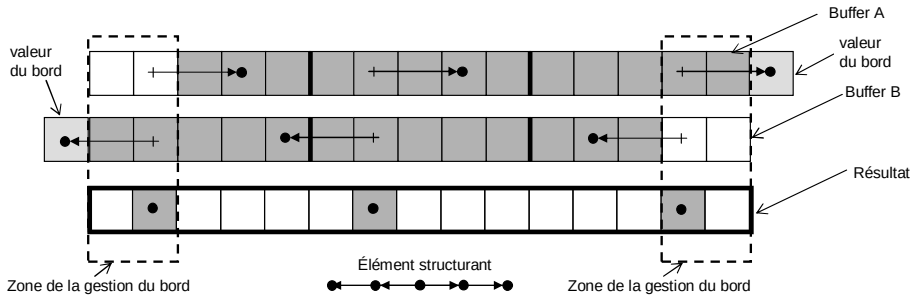


FIG. 8.6 : Exemple de la fusion des valeurs des buffers A et B lors de la phase p_3 pour un pixel dans la zone intérieure de l'image et pour les pixels dans les zones touchant les bords de l'image

```
phase3Gen :: [Char] → BroderFnc→ I → (α → α → α) → ( Ar (I,I) α, Ar (I,I) α) → Ar (I,I) α
phase3Gen dir brdfnc k f (bufA, bufB) =
                    listArray (bounds $ bufA)
                  ∘ (map ( λi → f ( sampGen brdfnc bufA (p1 i) )
                                  ( sampGen brdfnc bufB (p2 i) ) ))
                    ∘ indices
                    $ bufA
  where
    p1 (x1,x2) = if (how == "Fst") then (x1 + div (k−1) 2, x2) else (x1, x2 + div (k−1) 2)
    p2 (x1,x2) = if (how == "Fst") then (x1 − (div (k−1) 2), x2) else (x1, x2 − (div (k−1) 2))
```

Le paramètre dir spécifie la direction du segment ($"Fst"$ ou $"Snd"$) et la fonction $brdfnc$ assure l'extraction des valeurs pour les index à l'extérieur du domaine des deux buffers. k est le nombre des éléments constituant le segment et f est la fonction à effectuer (max ou min). Dans le cœur de cette fonction, nous créons le stream des index ixs par la fonction **indices** du Haskell, nous procédons à l'échantillonnage des deux buffers à l'aide de la fonction sampGen pour les positions décalées p_1 et p_2 et nous appliquons ensuite la fonction f. La fonction listArray crée, d'une manière standard, un array à partir d'un stream des résultats. Ainsi, nous obtenons une ligne (colonne) des valeurs résultantes de l'opération morphologique.

L'algorithme entier de van Herk-Gil-Werman que nous construisons à partir de ces trois phases et qui travaille avec les segments suivant l'axe de la première coordonnée (Fst) est présenté par l'algorithme 8.2 en tant que fonction algoHGWFst. Notre intérêt pour cette première coordonnée sera éclairci par la suite lors de la description de la version SIMD de cet algorithme. Notons que la spécification pour les segments suivant la direction de la deuxième coordonnée est analogue et peut être facilement dérivée à partir de la définition de la fonction algoHGWFst.

L'algorithme HGW prend quatre arguments : k est la taille du segment symétrique, $brdfnc$ est la fonction qui assure l'obtention de la valeur de l'extérieur du domaine de l'image lors d'échantillonnage des buffers dans la phase p_3, f est la fonction qui assure l'opération morphologique et ar est l'array d'entrée. Malgré la forme peu standard de cet algorithme, son fonctionnement n'est pas plus compliqué que ceux déjà rencontrés. Nous commençons par appliquer (sur la ligne 18) sur l'array ar le découpage, ligne 17, en colonnes composées des éléments de base. Nous passons, ligne 16, d'un array à un flux de données (colonnes dans ce cas) et nous appliquons sur chaque colonne (ligne 6) un bloc de traitement (entre les lignes 7 et 14).

Ce bloc fonctionnel est dédié à la construction des buffer A et B. Il ne fait que découper chaque colonne (ligne 14) en macro blocs de taille k du segment, exprime cette colonne en tant que flux des macro blocs et applique sur chacun d'eux la phase1 et la phase2 de traitement par la propagation des

Algorithme 8.2 : algoHGWFst, Algorithme de van Herk-Gil-Werman de dilatation/érosion par un segment suivant la direction de la première coordonnée

```
1  algoHGWFst  ::  I → BroderFnc→ (α → α → α) → Ar (I,I) α → Ar (I,I) α
2  algoHGWFst k brdfnc f ar =
3            arrayFromMxNBlocs
4          ∘ (listArray ((1,1),(1,q)))
5          ∘ (map (phase3Gen "Fst" brdfnc k f))
6          ∘ (map (
7                  ( λ(mbsA,mbsB) →(
8                        arrayFromMxNBlocs∘ ( listArray ((1,1),(mbnum,1)) )$ mbsA,
9                        arrayFromMxNBlocs∘ ( listArray ((1,1),(mbnum,1)) )$ mbsB
10                     )
11                 )
12                 ∘ ( λmbs→ (map (phase1 "Fst" f) mbs),( map (phase2 "Fst" f) mbs) )
13                 ∘ elems
14                 ∘ (arrayToMxNBlocs mbnum 1) )
15         )
16         ∘ elems
17         ∘ (arrayToMxNBlocs 1 q)
18         $ ar
19     where
20         (p,q) = dimsAr2D ar
21         mbnum= div p k
```

valeurs. Les lignes restantes de ce bloc fonctionnel (ligne 7 à 11) construisent deux buffers à partir des résultats du traitement à l'intérieur des macro blocs.

L'application de la phase3Gen se poursuit, ligne 5, à l'échelle du stream des colonnes et prend comme argument le stream des tuples composés, pour chacune des colonnes des deux buffers correspondants. Les lignes restantes (ligne 3 et 4) recomposent l'array de sortie à partir du stream résultant des colonnes.

8.2.2 Parallélisation pour les architectures SIMD

Une fois expliqué l'algorithme HGW pour les éléments de base, nous nous demandons de quelle manière nous pouvons explorer les capacités SIMD de notre architecture multimédia pour accélérer le calcul de cet algorithme. Puisque cet algorithme utilise les principes de traitement qui sont compatibles avec le traitement des données paquetées, nous pouvons facilement dériver un algorithme SIMD à partir de la description de l'algorithme travaillant avec les éléments de base, comme présenté par l'algorithme 8.2. Nous allons travailler, comme à l'échelle des colonnes, tant à l'échelle des macro blocs, avec les structures composées des vecteurs paquetés PVec I α plutôt que d'utiliser les structures composées des éléments de base du type α.

Formellement, nous pouvons décrire ce procédé par la spécialisation de l'algorithme 8.2 pour les données paquetées et l'algorithme qui décrira ce procédé devra assurer le paquetage de l'array d'entrée, appel de l'algorithme algoHGWFst et, à la sortie, également le dépaquetage des résultats afin d'obtenir l'array composé des données du type de base.

Cependant, nous ne pouvons pas choisir n'importe quelle direction pour la propagation des valeurs SIMD car nous sommes contraints par le sens de stockage des données vectorielles dans la mémoire. Ainsi, il est simple de construire l'algorithme pour les segments qui sont perpendiculaires au sens de stockage. En revanche, il est impossible d'utiliser l'algorithme SIMD de van Herk-Gil-Werman pour un segment parallèle au sens du stockage et nous sommes obligés d'effectuer une opération de changement du sens de stockage des données vectorielles, comme décrit dans le chapitre 6 dédié à cette problé-

matique. Dans ce cas, nous allons faire appel à la transposition des images entières. Remarquons que nous pourrions construire également des algorithmes qui exploiteraient l'approche des macro blocs et la transposition directe, mais nous soulignons que leur construction, même possible, n'est pas triviale.

C'est à cette place que nous allons expliquer pourquoi nous avons choisi l'axe "Fst" pour la construction de l'algorithme 8.2, défini par la fonction algoHGWFst. Supposant que les données sont stockées dans la direction de la deuxième coordonnée, ce sont les segments suivant la direction "Fst" pour lesquels nous pouvons construire l'algorithme SIMD d'une manière simple, comme présenté par l'algorithme 8.3 (défini par la fonction algoHGWFstSIMD).

Algorithme 8.3 : algoHGWFstSIMD, Algorithme SIMD de van Herk-Gil-Werman pour les segments suivant la direction de la première coordonnée (Fst) et en supposant que l'axe de stockage des données dans la mémoire est "Snd"

```
1 algoHGWFstSIMD :: I → I → BroderFnc → (α → α → α) → Ar (I,I) α → Ar (I,I) α
2 algoHGWFstSIMD n k brdfnc f ar =
3             mkAr2DFromAr2DPVecBySnd
4           ∘ (algoHGWFst k brdfnc f)
5           ∘ (mkAr2DPVecBySnd n)
6           $ ar
```

La construction de l'algorithme 8.4 (défini par la fonction algoHGWSndSIMD) exploitant les capacités SIMD pour les segments suivant la direction parallèle à l'axe de stockage de données est enrichie par la transposition par diagonale, exécutée par la fonction trRot2DMBSIMD, qui assure le changement du sens de stockage des données. Cela à deux reprises, avant et après l'exécution de l'algorithme algoHGWFstSIMD.

Algorithme 8.4 : algoHGWSndSIMD, Algorithme SIMD de van Herk-Gil-Werman pour les segments suivant la direction de la deuxième coordonnée (Snd) et en supposant que l'axe de stockage des données dans la mémoire est "Snd"

```
1 algoHGWSndSIMD :: I → I → I → BroderFnc → (α → α → α) → Ar (I,I) α → Ar (I,I) α
2 algoHGWSndSIMD mbsize n k brdfnc f ar =
3             (trRot2DMBSIMD "TD" "Snd" mbsize)
4           ∘ (algoHGWFstSIMD n k brdfnc f)
5           ∘ (trRot2DMBSIMD "TD" "Snd" mbsize)
6           $ ar
```

8.3 Résultats expérimentaux

Nous avons implémenté la version SIMD de l'algorithme de van Herk-Gil-Werman à l'aide des templates du langage C++, fournis avec le compilateur Intel ICL que nous avons utilisé pour cette implémentation. Pour pouvoir illustrer ses performances sur une architecture grand public, nous avons effectué d'autres tests qui ont une valeur informative car leurs temps d'exécution sont de loin moindres. La table 8.1 mentionne les résultats en chiffres. L'implémentation SIMD y est mentionnée en tant que *C++ template SSE2*.

Nous avons testé l'implémentation itérative, cf. la section 8.1, page 166, qui utilise à l'interne, comme l'algorithme de base pour les itérations, une implémentation optimisée manuellement au niveau d'instructions d'assembleur pour l'architecture Intel IA-32 qui exploitait l'exécution SIMD à l'échelle des registres 32 bits.

Nous mentionnons également deux autres implémentations qui ont été écrites dans le langage C, sans avoir exploré les capacités SIMD. La première (mentionnée comme *C non-SIMD** dans la tab. 8.1) a suivi directement la définition, utilisait abondamment les structures if-else dans les tests de dépassement des bords, et correspond à une implémentation intuitive qu'un programmeur effectue si la performance de l'algorithme n'est pas sa priorité. Nous avons restructuré le code de cette implémentation dans une forme plus propre qui ne fait plus appel aux instructions spécifiques à l'architecture (mentionnée comme *C non-SIMD***). Nous avons espéré que le compilateur pourrait, dans un tel code, procéder à la vectorisation de la même manière que nous l'avons exposée dans l'algorithme 8.3, mais malgré le gain que nous avons obtenus et qui est dû à la restructuration de cette implémentation, le temps d'exéctution est plutôt décevant, surtout si on le compare avec les temps obtenu pour les implémentations *C++ template SSE2* qui intègrent le travail SIMD dans la structure de l'algorithme.

Temps d'exécution en ms des algorithmes de la dilatation par segments								
algorithme	méthode	type	Taille du segment symétrique (1 $\sim$ 3 pixels)					
			1	3	5	15	30	60
Itératif	assembleur 32 bits	HOR	1.486	3.893	5.791	13.00	21.96	36.85
Itératif	assembleur 32 bits	VER	3.256	8.596	10.31	31.18	52.35	91.84
HGW	C non-SIMD*	HOR	—	—	31	—	—	—
HGW	C non-SIMD**	HOR	—	—	6.1	—	—	—
HGW	C++ template SSE2	HOR	0.626	0.625	0.622	0.602	0.602	0.592
HGW	C++ template SSE2	VER	0.334	0.331	0.313	0.276	0.261	0.260

Légende : HOR = segment horizontal, VER = segment vertical ; L'implémentation *assembleur 32 bits* est programmée en assembleur et utilise directement les instructions 32 bits de l'architecture Intel IA-32 ; l'implémentation *C non-SIMD** selon la définition mathématique, utilise abondamment les constructions if-else ; l'implémentation *C non-SIMD*** est la plus optimisée possible en C et à la main sans utiliser les types vectoriels. L'implémentation *C++ template SSE2* utilise les classes fournies avec le compilateur Intel ICL pour le calcul vectoriel. L'image d'entrée/sortie : $768 \times 576 \times 8\ bits$ = 432 ko ; processeur Intel Pentium 4 à 2.4 GHz, mémoire cache L2 = 512 ko ; système d'exploitation Microsoft Windows XP ; compilateur Intel ICL 7.1 pour Windows.

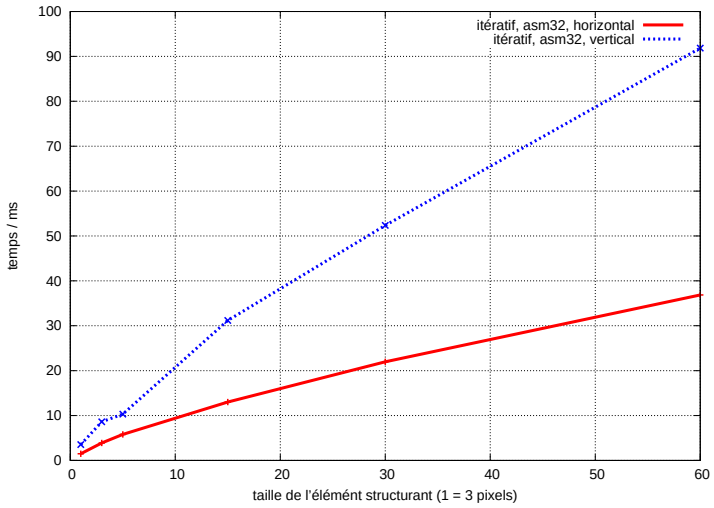
TAB. 8.1 : Résultats expérimentaux de diverses implémentations de la dilatation par segments

Le graphe présentant les temps d'exécution dépendant de la taille du segment pour l'implémentation itérative, cf. la fig. 8.7(a), a tout-à-fait la forme attendue sachant que la complexité de cet algorithme pour une image donnée est de $O(K)$, où K est le nombre de voisins traités par pixel. Les temps d'exécution grandissent avec la taille de l'élément structurant linéairement.

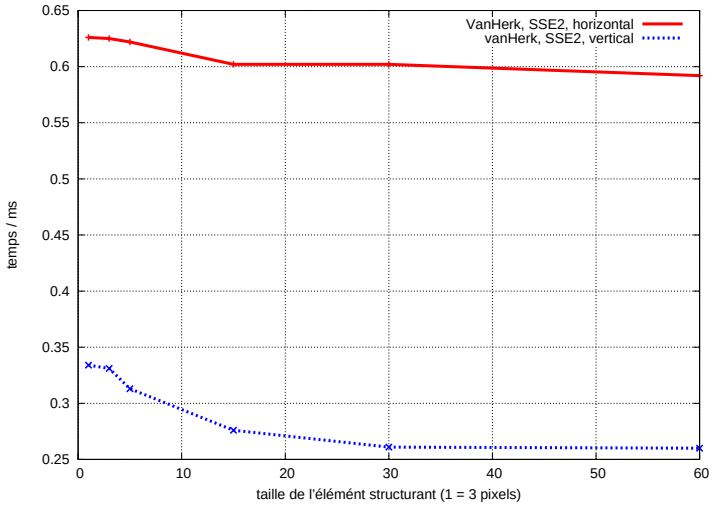
Nous présentons cette courbe comme contre-exemple pour démontrer qu'il est possible de calculer les dilatations / érosions beaucoup plus rapidement et surtout, avec un algorithme ayant la complexité $O(1)$, où le temps du calcul ne dépend pas du nombre des voisins traités par pixel, q.v. la fig. 8.7(b). Il s'agit de l'implémentation SIMD de l'algorithme de van Herk-Gil-Werman, cf. les définitions de la fonction algoHGWFstSIMD (l'algorithme 8.4) et de la fonction algoHGWSndSIMD (l'algorithme 8.4).

L'écart entre les deux courbes dans la fig. 8.7(b) qui est d'une valeur constante n'est pas surprenant car nous savons comment nous avons construit ces deux implémentations. Cet écart correspond, en effet, à 2 exécutions de la transposition par diagonale de l'image entière. Un phénomène plus intéressant présenté par les mêmes données est celui de la baisse du temps du calcul avec la taille du segment grandissant qui est, au premier regard, paradoxal.

Au début, nous avons jugé que cette baisse était due aux optimisations que le compilateur peut effectuer dans le code de boucles lors de l'exécution de la phase p_1 et p_2 à l'échelle des macro blocs. Mais l'explication est, en fait, plus simple. Il s'agirait d'un phénomène dû à la gestion des boucles car pour les tailles grandissantes des segments, nous obtenons moins de macro blocs et par conséquent, moins de boucles. Ce phénomène est présent également dans les implémentations non-SIMD de l'algorithme de van Herk et même dans d'autres, nous renvoyons le lecteur aux articles[DB05] et aux publications[DD06] qui présentent les graphiques avec des tendances similaires.



(a) Algorithme itératif



(b) Algorithme SIMD de van Herk-Gil-Werman pour Intel SSE2

FIG. 8.7 : Les temps du calcul des implémentations de la dilatation / érosion par un segment pour une image $768 \times 576 \times 8\ bits$ = 432 ko

8.4 Récapitulation

Ce chapitre a été consacré à l'explication des principes de la construction des algorithmes SIMD de la dilatation / érosion par les segments. Nous avons présenté le cas spécifique de l'implémentation de l'algorithme de van Herk-Gil-Werman pour les éléments structurants de la forme des segments horizontaux et verticaux. Cet algorithme nous a également servi dans cette thèse comme exemple d'un algorithme complexe qui réutilise toutes les techniques de travail avec les vecteurs paquetés présentées dans les chapitre précédents, notamment la gestion de la dépendance de l'exécution sur le parcours lors de la propagation des valeurs, l'exploitation de la non-dépendance de l'exécution sur le sens du parcours à l'échelle des macro blocs et, enfin, l'utilisation de la transposition par diagonale afin d'assurer le changement de l'axe de stockage des données.

Ces principes peuvent être réutilisés dans d'autres implémentations des algorithmes de ce type. Nous pensons, en particulier, aux implémentations sur la grille hexagonale, à l'implémentation de l'algorithme de van Herk-Gil-Werman avec la transposition directe par macro blocs et à l'implémentation SIMD des dilatations / érosions pour les segments inclinés qui n'est pas triviale car elle devrait gérer l'appartenance des données utilisées dans l'algorithme HGW à plusieurs macro blocs dédiés à la transposition. L'emploi de ces techniques dans les algorithmes pour les ouvertures et fermetures rapides par les segments est également à envisager.

CHAPITRE 9

Algorithmes et complexité

9.1 Définition d'un algorithme

Dans le contexte moderne, les *algorithmes* sont les méthodes pour la résolution des problèmes sur les ordinateurs. Dans le contexte plus large que celui restreint à l'informatique, nous comprenons sous le terme *algorithme* une prescription qui peut avoir différentes formes (textuelle, d'une formule mathématique, d'un diagramme de séquence, d'un langage de programmation, etc.). Cette prescription décrit une méthode de résolution d'un problème qui, après être effectuée, nous fournit la solution du problème.

L'utilisation des algorithmes n'est pas un phénomène de l'époque des ordinateurs et du calcul automatique. Les algorithmes ont été connus et employés depuis bien longtemps et cette utilisation remonte, d'après [Wik06a] Donald Knuth, à travers l'Antiquité jusqu'au Babyloniens (1800 avant J.C). En revanche, l'origine du mot *algorithme* est plus récent (9ème siècle) ; il a ses racines dans la traduction latine datant du Moyen Âge du nom de Abou Jafar Muhammad Ibn Musa al-Khuwarizmi [Wik06a] où son nom *al-Khuwarizmi*[1] était traduit en *Algoritmi*.

Par curiosité, nous pouvons découvrir [Wik06b] également que le titre *Al-jabr wa'l-muqabalah* d'un des ouvrages d'al-Khuwarizmi (publié en 825) qui signifie *La transposition et la réduction* est à l'origine d'un autre mot que l'on utilise fréquemment de nos jours – *algèbre*. Plus précisément, c'est sa partie *Al-jabr* qui après la traduction latine puis la traduction française devint *algèbre* et désigne aujourd'hui une discipline de la mathématique moderne dont certaines techniques sont déjà décrites dans l'ouvrage concerné d'al-Khuwarizmi.

9.2 Complexité d'un algorithme

9.2.1 Définition de la complexité

Sous le terme de *complexité* d'un algorithme donné nous comprenons le coût de l'utilisation de cet algorithme. Ce coût peut être exprimé dans diverses unités, par exemple, comme le temps du calcul, comme de nombre d'opérations arithmétiques, le nombre d'opérations avec la mémoire, etc.

9.2.2 Les mesures de la croissance

Même si, d'un point de vue pratique, le temps du calcul est une mesure extrêmement intéressante, elle n'est pas utilisable lors de l'étude théorique de la complexité des algorithmes où nous avons besoin de comparer le fonctionnement théorique de deux algorithmes différents et nous avons besoin de rester

[1] Appelé aussi al-Khwarizmi ou al-Khorezmi selon le nom de sa ville natale Khwarizm ou Khorezm ; à notre époque il s'agit de la ville Khiva en Ouzbékistan

abstraits d'une architecture particulière. C'est pourquoi nous faisons appel aux mesures qui définissent la relation entre deux fonctions mathématiques. Nous avons recours aux fonctions plutôt qu'aux numéros précis, car, dans la plupart des cas, c'est l'évolution du coût pour un ou plusieurs paramètres donnés qui nous intéresse.

Les mesures de croissance largement utilisées et décrites par les symboles o (petit o), O (grand o), Θ, $\sim$ et Ω sont dues à Donald Knuth[Wik06e]. Ces mesures sont toujours relatives à deux fonctions. Lors de la description d'un algorithme, nous allons substituer sa complexité par une de ces fonctions et nous allons modéliser cette complexité en choisissant convenablement une autre fonction qui va nous servir comme étalon tout en respectant la définition de la mesure de croissance choisie.

Les définitions des mesures de croissance que nous présentons sont extraites du livre traitant des algorithmes et de la complexité de Herbert S. Wilf[Wil94]. Pourtant, nous n'allons avoir recours qu'à certaines de ces fonctions pour exprimer la complexité des algorithmes dans cette thèse, notamment O et Ω qui semblent les plus adaptés à nos besoins.

Définition 9.1 (Définition d'o (petit o))**.**
Nous disons que $f(x) = o(g(x))$ *pour* $x \to \infty$ *si* $lim_{x\to\infty} f(x)/g(x)$ *existe et est égale à 0.*

Définition 9.2 (Définition d'O (grand o))**.**
Nous disons que $f(x) = O(g(x))$ *pour* $x \to \infty$ *si* $\exists C, x_0$ *tels que* $|f(x)| < Cg(x), \forall x > x_0$.

Définition 9.3 (Définition de Θ)**.**
Nous disons que $f(x) = \Theta(g(x))$ *s'il existe les constantes* $c_1 > 0, c_2 > 0, x_0$ *telles que* $c_1 g(x) < f(x) < c_2 g(x)$, *pour* $\forall x > x_0$.

Définition 9.4 (Définition de $\sim$)**.**
Nous disons que $f(x) \sim g(x)$ *si* $lim_{x\to\infty} f(x)/g(x) = 1$.

Définition 9.5 (Définition d'Ω)**.**
Nous disons que $f(x) = \Omega(g(x))$ *si* $\exists \epsilon > 0$ *est une séquence* $x_1, x_2, x_3, \cdots \to \infty$ *telle que* $\forall j : |f(x_j)| > \epsilon g(x_j)$

9.3 Modélisation des performances

La mesure de croissance utilisée le plus souvent dans la littérature pour estimer le coût des algorithmes est O. L'impact de cette mesure pour l'évaluation de la complexité est intelligible. Elle nous permet de rester abstraits des détails de fonctionnement d'un algorithme et de le modéliser, tout en remplissant la définition, par une autre fonction qui est plus simple à décrire et à comprendre. Elle est très utile dans le cas où nous comparons la complexité de deux algorithmes différents ; par exemple, il est évident qu'un algorithme dont la complexité est de $O(N^2)$ sera beaucoup moins avantageux que celui dont la complexité est de $O(N\log_2 N)$, pour $N \in \mathbb{N}$, car nous remplaçons, dans ce dernier, une fonction polynomiale par une fonction contenant des logarithmes dont la croissance est moindre.

Pourtant, cette mesure n'est pas la meilleure pour l'estimation pratique du coût des algorithmes. En effet, ce n'est pas sa vocation principale car elle définit la borne asymptotique maximale. En se basant sur N, le nombre des éléments à traiter, cette mesure ne capte pas le vrai coût que nous devons payer pour l'évaluation de ces éléments.

Dans la morphologie pratique, comme dans d'autres domaines d'ailleurs, nous ne travaillons pas avec des images excessivement grandes, d'autant moins avec les images dont le nombre d'éléments approche l'infini. Ce fait est accentué lors du traitement d'images en temps réel où nous essayons de réduire le temps du traitement le plus possible et où nous travaillons avec les sections d'une image ou avec des images sous-échantillonnées. Dans ces cas, l'estimation de la complexité via $O()$ peut nous conduire au choix d'un algorithme inadapté à la situation particulière que nous rencontrons.

Expliquons ce fait sur un exemple synthétique mais qui démontre bien la situation. Ayons un premier algorithme dont la complexité est $O(N^2)$ et ayons un deuxième algorithme dont la complexité est

$O(N\log_2 N)$ où N est le nombre des éléments, $N \in \mathbb{N}$. En principe, nous sommes tentés d'utiliser le deuxième algorithme dont la complexité est exprimée par une fonction moins croissante. Cependant, nous ne pouvons pas comparer le coût exact par une simple comparaison de $O()$ des deux algorithmes. La complexité exprimée en $O()$ ne nous indique pas le coût exact pour un cas précis et nous pouvons effectivement trouver des cas spéciaux pour lesquels le choix du premier algorithme serait une solution plus adaptée.

Imaginons que le premier algorithme utilise pour évaluer les éléments les opérations de base, en revanche, le deuxième algorithme va utiliser des opérations plus complexes ; ce qui va se traduire par un coût plus important pour le traitement d'un élément du deuxième algorithme. Supposons que le rapport entre les coûts des deux algorithmes, cette fois-ci n'étant pas exprimé par le $O()$, est lié par une constante $c \geq 1$ et que nous cherchons les valeurs précises de N pour lesquelles le choix du deuxième algorithme est justifié. Nous essayons, en effet, de trouver la solution de l'inégalité :

$$N^2 \geq cN\log_2 N$$

ce qui peut être récrit comme :

$$N \geq c\log_2 N$$

ou encore :

$$2^N \geq N^c$$

Ici, nous comparons une fonction exponentielle avec une fonction polynomiale. Si nous essayons de concrétiser les valeurs, nous nous apercevons que pour $c = 1$ et $c = 2$, l'inégalité est valable $\forall N \in \mathbb{N}$ tant que pour $c \geq 3$, nous obtenons un intervalle (pour $c = 3$ il s'agit des valeurs $N = \{2, 3, 4, 5, 6, 7, 8, 9\}$) dans lequel l'inégalité n'est pas valable. Pour les valeurs N concrètes incluses dans cet intervalle, l'emploi du premier algorithme serait plus avantageux. Cet intervalle s'élargit avec la valeur c grandissante.

9.4 Estimation de la complexité et des performances pour les GPP/GPPMM

9.4.1 Idée générale

L'exemple décrit précédemment devait nous démontrer l'utilité pratique de la spécification précise lors de l'estimation de la complexité d'un algorithme. À cette occasion, l'utilisation de O ou de la mesure plus restrictive o n'est pas la meilleure solution.

Pour pouvoir exprimer le coût de nos algorithmes avec l'intention de pouvoir estimer le temps du calcul, nous voudrions identifier les divers paramètres qui influent le plus le coût d'un tel algorithme. Vu que nos algorithmes, même généralisés, seront fortement orientés sur les architectures GPP et sont surtout GPPMM qui ont un fonctionnement particulier, l'expression qui estimera la complexité pratique d'un algorithme peut être perçue comme la construction d'un modèle de performance de cet algorithme pour la classe des architectures GPPMM.

En pratique, nous allons distinguer explicitement les opérations qui ont un coût différent et nous allons les présenter comme telles dans notre modèle de performance. Les différences entre le modèle et la réalité sont toujours présentes et c'est pourquoi nous allons lier ce modèle théorique avec le vrai coût d'un algorithme par la mesure Θ, cf. sa définition, page 178. Cette mesure est capable d'assimiler, via les constantes c_1 et c_2 de sa définition, les différences entre notre modèle de performance et la performance exacte d'un algorithme.

Nous avons une importante remarque à faire à cette place concernant l'utilisation pratique de Θ comme nous venons de le présenter. Si $f(x) = \alpha x^2$ est la complexité exacte d'un algorithme, nous pouvons écrire $f(x) = \Theta(\alpha x^2)$. Mais même si le terme α peut désigner le coût précis par pixel, nous pouvons exprimer cette complexité également comme $f(x) = \Theta(x^2)$ sans s'attaquer à la validité de l'estimation. Le fait d'utiliser les termes constantes à la même place où nous avons utilisé α est critiqué (Wilf[Wil94], page 6) comme un mauvais style car α n'ajoute aucune information vis-à-vis la définition de

Θ. Pourtant, il nous semble utile de l'y mentionner car si ce terme α est bien décrit et reflète correctement la réalité, il peut nous démontrer plus en détail la structure de la fonction du coût d'un algorithme.

Notons que la seule manière de vérifier que notre estimation, que nous appelons *estimation pratique de performance*, est correcte et cohérente avec la réalité, c'est la mesure physique du temps du calcul d'un algorithme sur une architecture donnée et pour les paramètres donnés (les dimensions d'image, d'élément structurant, etc.).

9.4.2 Estimation pratique pour les GPPMM

Expliquons sur un exemple trivial comment nous envisageons de procéder à cette estimation pratique. Nous prenons l'exemple de l'opération addition de deux images (arrays) de dimensions $M \times N$ et dont le résultat serait également une image. Nous voulons exécuter cette opération sur une architecture GPP avec un seul processeur. La complexité C_1 de l'algorithme trivial travaillant élément par élément exprimé en O est $O(MN)$ ce qui peut également être exprimé, en effet, comme :

$$C_1 = O(N^2). \tag{9.1}$$

Si nous voulons décrire un modèle de performance précis, il faut d'abord distinguer les opérations avec la mémoire des opérations arithmétiques qui peuvent participer au coût final par différentes proportions. Notons comme μ le coût de toute les opérations avec la mémoire (lecture et écriture) pour un élément d'image. Notons comme α le coût de toutes les opérations arithmétiques qui sont nécessaires pour évaluer un élément. Ainsi, le coût de cet algorithme peut être estimé comme

$$C_1 = \Theta((\alpha + \mu)MN). \tag{9.2}$$

ce qui donne une idée plus précise de son fonctionnement.

Regardons comment vont changer ces estimations si nous utilisons à la place d'une architecture GPP avec un seul processeur (un seul fil d'exécution physique) une architecture GPPMM à plusieurs processeurs et/ou à plusieurs cœurs et/ou à plusieurs fils d'exécution indépendantes et/ou à parallélisme superscalaire avec les capacités SWAR. Notons par P le nombre de processeurs qui peuvent assurer l'exécution concurrente et que S soit le nombre d'éléments qui peuvent être traités par les instructions SIMD en même temps. Puisque les coûts d'accès à la mémoire et les coûts des opérations arithmétiques ne sont pas, dans le cas général, les mêmes que ceux présentés dans l'équation 9.2, nous allons dénoter par β le coût de toutes les opérations arithmétiques nécessaires pour l'évaluation d'un élément multimédia large et par ν le coût d'accès à la mémoire pour un tel élément. Pour cette configuration matérielle, il est possible de concevoir un algorithme dont la complexité C_2 sera :

$$C_2 = \Theta((\beta + \nu)\frac{MN}{SP}). \tag{9.3}$$

Pour pouvoir obtenir des estimations très concrètes, il faut substituer aux termes α, μ, β et ν des valeurs concrètes qui peuvent être exprimées comme multiples de cycles d'horloge de notre architecture. Pour faire cela, il faut très bien connaître l'architecture de notre matériel informatique et surtout les phénomènes entrant en jeu. Il s'agit souvent des effet indésirables du préchargement de données dans la mémoire cache en temps d'exécution ce qui se manifeste, pour les grandes images n'entrant pas entièrement en mémoire cache, par un ralentissement assez important de l'exécution. Ce phénomène est accentué dans le cas où la zone de la mémoire de sortie est distincte des deux zones de mémoires d'entrée car dans ce cas, nous travaillons effectivement avec 3 images et le volume des données traitées peut très rapidement dépasser la taille de la mémoire cache.

Ce point est d'autant plus marquant sur les fonctions triviales telles que l'addition que nous abordons ici. Ce type d'opérations n'exécute pas assez d'instructions arithmétiques pour pouvoir cacher la préparation des données dans l'exécution confluente des instructions en pipeline (cf. 3.2.3, page 38).

Pour illustrer cet exemple, nous présentons dans la table 9.1 les temps du calcul de l'opération addition avec saturation que nous avons obtenus comme des résultats expérimentaux.

dimensions d'image	volume de données manipulé	méthode d'implémentation	temps en ms	taux d'accélération
$128^2 \times 8$bits	3*16 ko = 48 ko	générique	0.175	0.26
		via pointer++	0.045	**1.0**
		multimédia 64 bit	0.005	9
$256^2 \times 8$bits	3*64 ko = 192 ko	générique	0.69	0.28
		via pointer++	0.19	**1.0**
		multimédia 64 bit	0.03	6.3
$512^2 \times 8$bits	3*256 ko = 768 ko	générique	3.00	0.2
		via pointer++	0.60	**1.0**
		multimédia 64 bits	0.40	1.5

Opération travaillant avec 3 images, chacune dans une zone de mémoire distincte. L'implémentation *générique* utilise les fonctions getPixel()/setPixel() ; l'implémentation *via pointer++* travaille élément par élément en utilisant les pointeurs ; l'implémentation *multimédia 64 bits* utilise les instructions spéciales SIMD de 64 bits pour évaluation. Machine : Intel Pentium 4 @ 2.4 GHz, single thread, 8 ko L1, 512 ko L2 cahce, Linux Mandrake 9.2. Compilateur Intel ICC 8.1, optimisations manuelles dans le cas *multimédia 64 bits*.

TAB. 9.1 : Temps du calcul de l'opération addition avec saturation sur les images dont les éléments sont du type unsigned integer 8bit

Il est évident que pour le volume de données traitées de 48 ko et 192 ko qui entrent dans la mémoire cache L2 de notre machine (512 ko), les taux d'accélération 9 et 6.3 sont cohérents avec la valeur attendue pour le travail SIMD de 64 bits qui devrait nous fournir, en théorie, le taux d'accélération égale à 8. En revanche, pour les images 512^2 dont la taille correspond aux volume manipulé de 768 ko, nous dépassons les capacités de notre architecture ; cela se reflète sur le temps du calcul qui dépasse les temps que l'on pouvait espérer en faisant une simple extrapolation des résultats précédents. Le taux d'accélération tombe dans ce cas à une valeur très décevante 1.5.

Cet exemple très pratique nous démontre que notre estimation devrait inclure également dans le coût pour les opérations avec la mémoire μ et ν un terme supplémentaire dont le comportement serait non-linéaire et dépendant de la taille de l'image et de la taille de la mémoire cache. Nous pouvons l'exprimer par la décomposition des coefficients du coût μ et ν comme :

$$\mu = \mu_1 + \mu_2(MN)$$
$$\nu = \nu_1 + \nu_2(MN)$$

où les premiers termes μ_1 et ν_1 expriment les coûts que l'on paye pour un accès aux données présentes dans la mémoire cache ; et où les deuxièmes termes μ_2 et ν_2, qui sont les fonctions de la taille de l'image MN expriment les coûts relatifs au comportement particulier lors du chargement de données dans la mémoire cache en cas de leur absence.

9.5 Exemple d'estimation pratique de la complexité des algorithmes de voisinage

En ce qui concerne la complexité de l'approche naïve, il ne suffit pas d'avoir que le skeleton algorithmique ngbAlgo pour son évaluation. Ce skeleton est très générique pour pouvoir effectuer l'étude de la complexité en ne se basant que sur lui. Il faut spécifier également les scénarios de son utilisation qui sont en directe correspondance avec les algorithmes concrets dilSQR et dilHEXR que nous venons de présenter.

L'approche naïve telle qu'elle est définie est générique. Elle utilise la fonction d'extraction du voisinage qui teste pour chaque accès au voisin si celui est inclus dans le domaine de l'image. Étant donné une image de $M \times N$ pixels, $N, M \in \mathbb{N}$, un élément structurant composé de K vecteurs, $K < MN$.

À partir de ces indices, nous pouvons estimer la complexité de l'approche naïve par O comme :

$$C_N = O(KMN) = O(KN^2) \tag{9.4}$$

Pour exprimer la complexité et sa structure plus en détail, nous devons spécifier d'autres paramètres cruciaux. Que $\alpha(K)$ désigne le coût du calcul des opérations arithmétiques ou logiques dans le kernel, ce coût et une fonction du nombre des éléments de l'élément structurant. Que L désigne le nombre d'accès qui devrait être effectués au-delà du domaine de l'image lors de l'extraction des voisins. Que τ désigne le coût que nous payons pour tester si l'index d'un élément est dans le domaine de l'image ou pas et que μ désigne le coût d'accès à un élément dans a mémoire et que π désigne le coût de l'obtention d'un élément au-delà des bords de l'image.

Lors du calcul d'un algorithme de la morphologie mathématique qui utilise le skeleton algorithmique ngbAlgo, la complexité pratique peut être exprimé comme :

$$C_N = \Theta(\alpha(K)MN + \tau KMN + \mu(KMN - L) + \pi L) \tag{9.5}$$

où le premier terme $\alpha(K)MN$ désigne le coût des opérations arithmétiques, le deuxième τKMN le coût des testes si l'élément à extraire est à l'intérieur du domaine, troisième $\mu(KMN - L)$ représente le coût d'accès à la mémoire pour les éléments qui sont à l'intérieur du domaine. Le quatrième πL représente un coût généralisé pour l'obtention des éléments qui sont à l'extérieur du domaine, ce terme peut exprimer aussi bien le coût d'un bord d'une valeur constante mais également le coût d'extraction d'un pixel du domaine de l'image si nous travaillons avec les bords qui reflètent le contenu de l'image. L'équation 9.5 peut être récrite comme :

$$C_N = \Theta((\tau + \mu)KMN + \alpha(K)MN + (\pi - \mu)L) \tag{9.6}$$

L'équation 9.6 nous démontre la structure des coûts de cette approche naïve. Les idées pour une amélioration des performances surgissent directement de cette équation et nous verrons par la suite comment nous pouvons réduire des coûts en changeant la structure de notre algorithme.

En ce qui concerne les stratégies de parallélisation de cette approche naïve, nous exploiterons les paradigmes de la parallélisation de données et la parallélisation des tâches. La forme exacte de parallélisation dépend des possibilités de notre matériel parmi lesquelles la réplication fonctionnelle exprimé par le skeleton farm est la plus simple est conduirait à un changement dans l'algorithme 5.1 dont nous présentons ici les lignes changées et où nous utilisons à la place de la fonction **map** la fonction farm :

```
4                    ∘ (farm op)
5                    ∘ (farm (extr ar))
```

Ce changement est mineur en ce qui concerne la forme de cet algorithme mais il introduit des conséquences majeures sur la conception de l'architecture et sur les performances. Vu que l'extraction du voisinage d'un pixel peut prendre un temps différent et le plus souvent plus long que le calcul de l'opération du kernel, les stratégies de parallélisation peuvent varier fortement. Pour le bon équilibre de la charge des blocs dans le pipeline d'exécution, nous pouvons choisir, dans le cas général, la multiplication des moyens matériels différente pour le kernel d'extraction des voisins et pour le kernel de l'opération morphologique.

Notons que d'exprimer la complexité d'une telle stratégie de parallélisation à partir de l'équation 9.4 à l'aide d'O posera des problèmes car cette formule ne décrit pas explicitement la complexité des différentes parties du pipeline d'exécution mais celle du pipeline tout entier. C'est pourquoi nous nous basons sur l'équation 9.6 de la complexité pratique où nous distinguons les opérations avec la mémoire des opérations arithmétiques. Si E est le nombre des unités qui sont dédiées à l'exécution en parallèle de la fonction d'extraction des voisins et P est le nombre de processeurs qui sont dédiés au calcul de l'opération sur le voisinage en parallèle, la complexité qui estime le temps du calcul pour la configuration

parallèle pourra être exprimée comme :

$$C'_N = \Theta(\frac{(\tau + \mu)KMN + (\pi - \mu)L}{E} + \frac{\alpha(K)MN}{P}) \tag{9.7}$$

Il est évident que nous obtenons les meilleures performances vis-à-vis du temps de calcul dans le cas où le traitement sur toutes les unités parallélisées de notre chaîne est équilibré.

9.6 Estimation de la complexité et des performances pour les GPU

Même s'il n'est pas très difficile de décrire la complexité des algorithmes pour les GPU à l'aide de $O()$ en se basant sur les éléments traités, la prévision des performances, comme nous le faisons pour les GPP à l'aide de $\Theta()$ n'est pas simple à effectuer.

C'est la combinaison des processeurs GPP - GPU qui rend cette estimation difficile. Dans cette combinaison, il s'agit d'une machine distribuée, avec tous les phénomènes qui sont propres au calcul distribué, dont les plus importants sont les délais dûs au transfert de données d'entrée et de sortie et les délais dûs au temps de synchronisation et au passage des commandes graphiques qui peuvent avoir une structure complexe et représenter un volume de données important.

Il faut également percevoir le GPU lui-même comme une structure de multiples processeurs, cf. fig. 3.18, page 52. Puisqu'il s'agit d'une structure architecturale qui est chaînée, la performance finale est fortement dépendante du type des algorithmes que nous effectuons et de la manière dont nous utilisons les unités exécutives de cette chaîne. Chacune de ces unités est constituée d'un matériel informatique spécialisé et est caractérisée par un niveau de parallélisation différent. Ce qui se traduit par des capacités de calcul différentes d'une unité à l'autre. Si une des unités est saturée par le traitement, tout le pipeline est saturé et on parle, dans le domaine de la programmation des GPU, du traitement *limité par* cette unité. Les capacités des ces unités sont optimisées par le fabricant pour les applications de la synthèse graphique, elles peuvent s'avérer non-optimales ou complètement inadaptées au traitement d'analyse d'images que nous visons dans cette thèse. L'optimisation des programmes pour obtenir un bon équilibre du calcul entre ces unités et pour augmenter ainsi la performance est même le sujet de nombreux articles que l'on peut trouver dans la littérature[CW02, Spi03].

9.6.1 Transfert de données

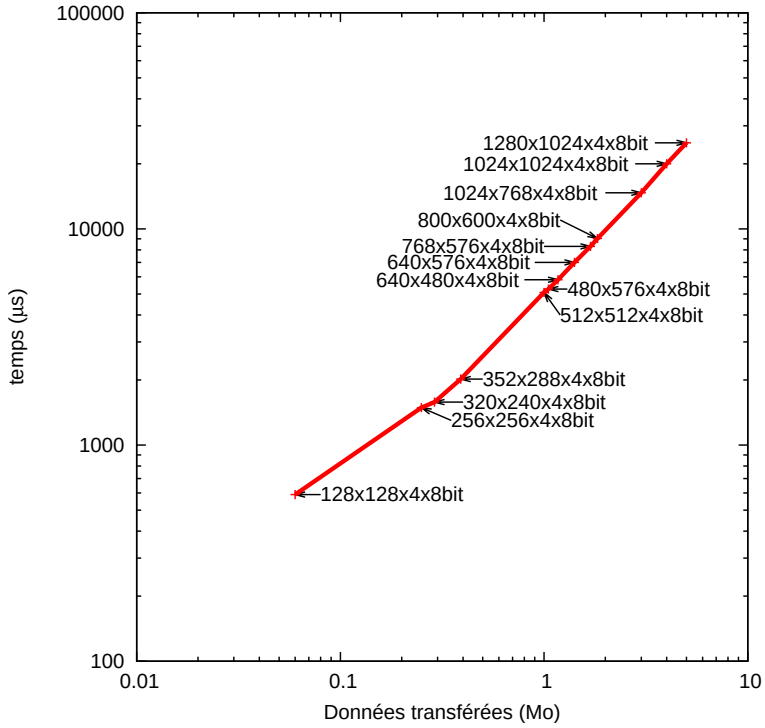
Le premier facteur très limitant pour notre traitement est représenté par les temps que nous perdons lors du transfert de données et cela dans les deux sens (GPP↔GPU). Notons que les articles traitant du transfert de données du point de vue de la programmation, un sujet particulier mais important, ont été déjà présentés et nous les recommandons au lecteur[Ake03].

Le bus AGP que nous avions à disposition est asymétrique. Le débit dans le sens GPP→GPU est supérieur (théoriquement jusqu'à 2.1 Go/s pour AGP 8x) à celui dans le sens opposé (théoriquement 266 Mo/s, ce qui est équivalent à "1x").

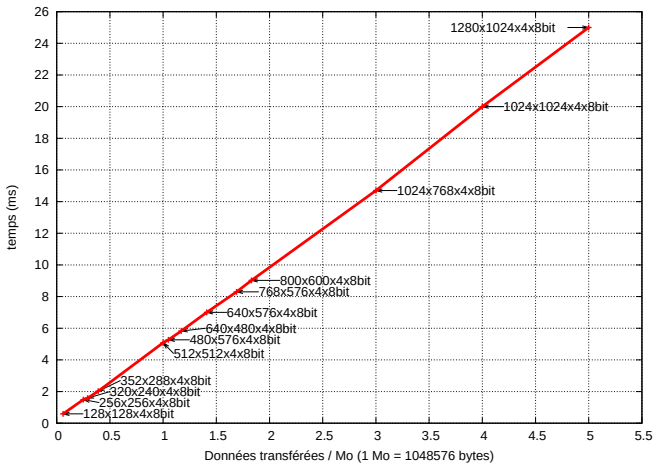
Nous présentons les temps obtenus expérimentalement pour le transfert de données GPU→GPP sur la fig. 9.1. Ces temps correspondent à AGP 1x et nous montrent les temps excessivement longs pour un travail en temps réel : par exemple, le transfert d'une image $256^2 \times 4$ bits prend 1.5 ms, ce qui est supérieur dans certains cas à la durée du traitement de ces données par le processeur graphique, comme nous l'avons pu voir dans les résultats expérimentaux du chapitre 5, page 123, notamment sur le temps d'exécution de 0.65 ms de la dilatation par un disque en 4-voisinage de taille 1 pour une image de 1 Mo.

Ces données représentent un aspect linéaire de croissance pour un volume de données transférées grandissant, cf. fig. 9.1(a), avec une déviation pour les images relativement petites ($128^2 \times 4$) qui est perceptible à échelle logarithmique, cf. fig. 9.1(a).

L'utilité du bus AGP pour le calcul GPGPU est discutable. Ce bus devint obsolète avec l'arrivée du nouveau bus PCI Express. Il s'agit d'un bus sériel (AGP était un bus parallèle), il est symétrique et plus

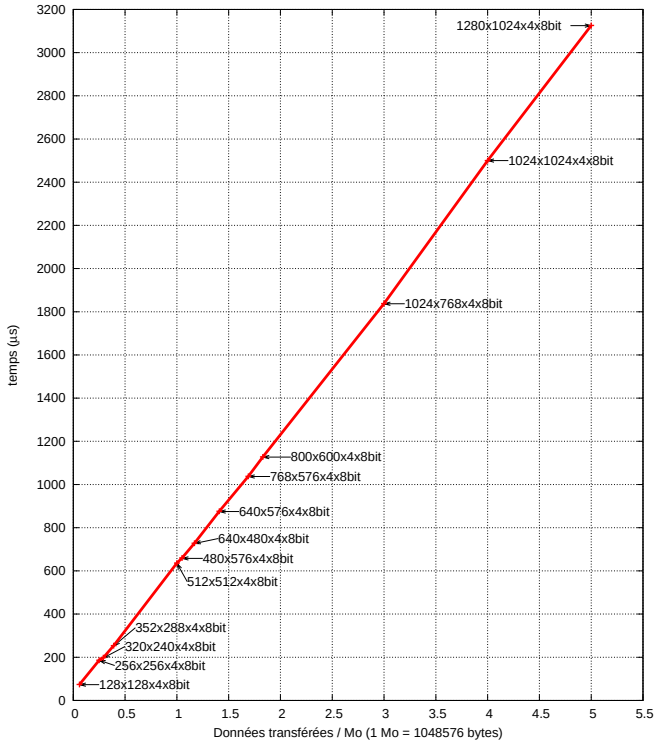


(a) Échelle logarithmique

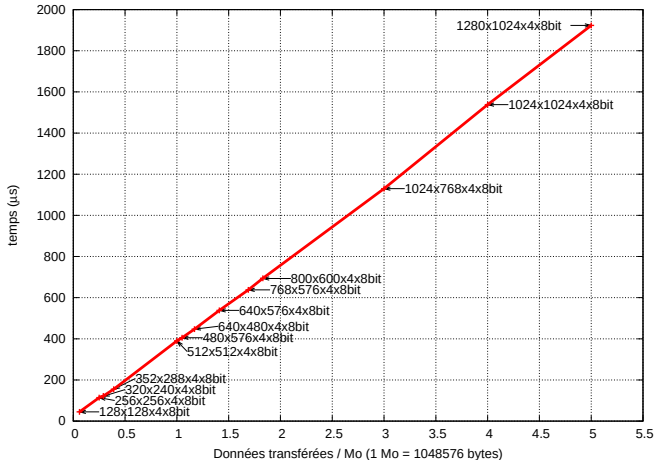


(b) Échelle linéaire

FIG. 9.1 : Temps du transfert, depuis GPU vers GPP, AGP (1x, débit théorique maximal 266 Mo/s). GPU = NVidia GeForce 6800 LE, GPP = Intel Pentium 4 @ 2.4 GHz 512 Mo RAM



(a) NVidia GeForce 6800 Ultra, taux effectif = 8x



(b) NVidia Quadro FX 4500, taux effectif = 13x

FIG. 9.2 : Temps du transfert estimé depuis GPP vers GPU, pour PCI Express (16x) et les textures fixed-point 8 bit BGRA (basé sur les données officielles de NVidia[NVi05]). GPP = AMD Athlon 64 bit 3500+, 1 Go RAM

Remplissage de rectangle	volume traité	Windows XP / fps		Linux Mandrake / fps	
		Fenêtre	Plein écran	Fenêtre	Plein écran
couleur constante	4 Mo	1670	2700	1422	3300
	3 Mo	—	3836	1850	3915
couleur variante	4 Mo	—	1500	1031	1663
	3 Mo	—	2326	1338	2190

4 Mo$= 1024^2 \times 4 \times 8$ bits, 3 Mo$= 1024 \times 768 \times 4 \times 8$ bits ; GPP = Intel Pentium 4 à 2.4 GHz single thread ; GPU = NVidia GeForce 6800 LE sur AGP 4x. Résolution d'écran lors du travail avec *fenêtre* = 1280×1024

TAB. 9.2 : Test comparatif de performances d'affichage d'un rectangle couvrant entièrement la scène

rapide, son débit théorique est de 4 Go/s dans sa version "16x", couramment présente dans l'année 2006, est deux fois supérieur à celui de AGP 8x dans le sens GPP→GPU et 16 fois supérieur dans le sens GPU→GPP.

Le bus PCI Express devait améliorer les temps du transfert, principalement dans le sens GPU→GPP. Même si une amélioration est perceptible, l'impact des transferts reste toujours important vis-à-vis de la durée du traitement GPGPU que nous effectuons.

De plus, on a beau avoir un débit théorique chiffré, les temps de transfert réels peuvent en différer fortement. Le temps de transfert varie, en effet, selon la configuration de notre architecture GPP-GPU. En se basant sur les données officielles de NVidia, nous avons modifié l'échelle de la courbe que l'on vient de présenter sur la fig. 9.1(b) pour pouvoir démontrer les temps estimés à partir de l'article[NVi05] qui sont les plus favorables à notre traitement (texture fixed point de 8 bit en format BGRA) pour deux processeurs graphiques différents. Sur la fig. 9.2(a), nous présentons les temps estimés de transferts GPP→GPU pour le processeur graphique NVidia GeForce 6800 Ultra connecté via PCI Express 16x et d'où nous pouvons déduire le taux effectif de "8x". Sur la fig. 9.2(b) nous présentons les temps estimés de transferts GPP→GPU pour le processeur graphique NVidia Quadro FX 4500 connecté également via PCI Express 16x et d'où nous pouvons déduire le taux effectif de "13x".

9.6.2 Influence du système d'exploitation et de l'API

L'influence non négligeable sur les performances finales est apportée également par le système d'exploitation à l'aide duquel nous exécutons nos programmes (dans notre cas Linux Mandrake 9.2 et Microsoft Windows XP) et, bien sûr, du pilote qui exécute nos commandes OpenGL ou DirectX.

En utilisant l'API Mesa sous Linux et l'API DirectX 9 sous Microsoft Windows XP, nous avons effectué un test comparatif dont les résultats sont présentés dans la tab. 9.2. Nous avons implémenté deux algorithmes d'affichage d'un rectangle couvrant entièrement la scène.

L'algorithme plus simple, dénoté *couleur constante*, n'utilise pas l'information de couleur incorporée dans les vertex, celle-ci est fournie par le programme traitant des fragments. Les résultats de cet algorithme nous servent comme indicateur de performance maximale atteignable de notre architecture GPP-GPU. Dans ce cas précis, nous effectuons le moins de travail dans le pipeline graphique. La valeur que nous allons observer sera le nombre de trames que nous pouvons traiter par seconde (fps).

Le deuxième algorithme, dénoté *couleur variante*, utilise l'information de couleur dans chacun des vertex du rectangle et cette information est pour chacun des vertex différente. Cette configuration occupe principalement le rastériseur et devrait nous démontrer le changement du nombre des trames traitables par seconde lors d'interpolation de 4 valeurs de couleur.

Puisque dans la plupart de cas, lors de notre travail avec les GPU, nous passons comme commandes graphiques les structures ayant un nombre très petit de triangles, les mesures expérimentales devaient nous tester également la validité de l'hypothèse que nous nous avons construite pendant l'étude de la bibliographie .

Il s'agit de démontrer expérimentalement que l'API basé sur OpenGL gère plus efficacement l'en-

voi de commandes qui comptent peu de triangles, cet fait étant mentionné dans la présentation de Wloka [Wlo03]. Le test que nous effectuons ici semble convenable pour cette démonstration. Nous y envoyons 1 rectangle (qui est constitué de 2 triangles), il suffirait de comparer les fps obtenus pour les deux systèmes d'exploitation.

Nous remarquons que ce type de travail n'est pas propre à l'utilisation habituelle des GPU car ces derniers ne sont pas conçus pour un travail à une fréquence élevée d'affichage dans le framebuffer mais ils sont conçus pour les algorithmes qui se présentent par des taux des fps moindres à 100, le nombre propre à la fréquence de rafraîchissement des trames pour la visualisation.

Les résultats nous révèlent trois idées principales :

- Il est à recommander de travailler avec une application qui fonctionne en régime *plein écran* (cf. les colonnes *Plein écran* dans la tab. 9.2). Dans le cas opposé où nous travaillerons dans une fenêtre de système d'exploitation (ou de système de fenêtres dans le cas de Linux), les taux de fps chuteraient d'une manière significative (cf. les colonnes *Fenêtre* dans la tab. 9.2).
- On peut percevoir une certaine supériorité des résultats pour Linux et le pilote du processeur pour OpenGL en terme de fps qui sont plus grandes que celles pour Windows et DirectX. Mais nous trouvons également le cas contraire, notamment les valeurs de fps pour les images de 3 Mo lors d'un remplissage par la couleur variante en *plein écran*. Il faut souligner que les résultats des deux API ne sont pas excessivement différents et nous ne pouvons pas faire une recommandation d'utilisation d'un API plutôt que de l'autre. De plus, ces résultats étant fortement dépendants de la version du pilote, ils peuvent varier fortement d'une version à l'autre.
- Le fait d'avoir ajouté au traitement de base l'interpolation des couleurs se manifeste par une diminution des fps allant dans certains cas jusqu'à 2 fois (Linux, *Plein écran*, 4 Mo). Sachant que ce test devait démontrer les capacités d'interpolation des valeurs (nous visons prioritairement l'interpolation des coordonnées des index de textures lors du travail avec 4 textures, sans avoir effectué l'échantillonnage), les résultats sont plutôt décevants car lors du travail avec les algorithmes de traitement d'images, nous allons encore ajouter aux temps qui correspondent à ces fps la durée des opérations de traitement des fragments (échantillonnage, opérations arithmétiques, etc.).

9.7 Récapitulation

Ce petit chapitre devait nous servir à introduire des techniques pour la description de la complexité des algorithmes. Nous y avons présenté, sur un exemple pratique pour les GPP/GPPMM les phénomènes que nous devons assumer si nous voulons passer de l'expression théorique de la complexité à l'aide de $O()$ à une estimation pratique du coût d'un algorithme, qui est, dans notre cas, prioritairement lié à l'expression du temps du calcul.

Cette estimation pratique n'est pas simple et est fortement liée à une architecture donnée mais également, comme nous l'avons présenté dans 9.4.2, page 180, à d'autres facteurs qui ne sont pas a priori aussi évident à assumer, comme les dimensions relatives d'une image et de la mémoire cache de l'architecture.

L'estimation pratique des performances des algorithmes pour les GPU est encore plus délicate. Pour pouvoir obtenir des estimations fiables, on devrait prendre en compte beaucoup de paramètres. De ces derniers, nous avons présenté en particulier le temps de transfert de données et la possible influence du système d'exploitation et de l'interface de programmation (l'API) que nous utilisons pour exécuter nos commandes graphiques. Remarquons que pour un programme donné et pour un processeur graphique donné, les outils de développement de NVidia permettent de calculer avec précision le nombre de cycles que les programmes pour les vertex et pour les fragments sont censés consommer. Mais il ne s'agit que de deux des unités du pipeline graphique et il n'existe pas une méthode pour pré-évaluer les performances du pipeline graphique dans son entier.

Même dans la littérature portant sur le GPGPU, nul ne présente les estimations de la performance pour les algorithmes pour les GPU. C'est dû au caractère des données, dépendantes de l'interaction avec l'utilisateur, qui sont traitées dans la synthèse d'images. Par conséquent, il n'est pas possible de prévoir

le nombre de triangles qui serait nécessaires au traitement. Il est beaucoup plus fiable et intéressant d'un point de vue pratique d'effectuer un test concret, sur un matériel donné.

Ce n'est pas la situation de l'analyse d'image où, pour la plupart des traitements, le nombre de données à traiter est prévisible avant l'exécution. C'est justement dans le cadre de GPGPU où une estimation plus précise des performances d'un algorithme serait la bienvenue et possible à réaliser. Nous laissons cette idée comme un problème ouvert qui pourrait conduire à une possible construction d'un modèle de performance crédible ; d'un modèle à partir duquel nous pourrions estimer le coût total d'un algorithme GPGPU, sans avoir besoin de faire appel aux tests expérimentaux.

Conclusion et perspectives

Cette page est blanche par intention

Conclusion et perspectives

Conclusion générale

Idées principales exposées dans la thèse

Cette thèse était consacrée aux algorithmes de la morphologie mathématique qui perçoivent les données en tant que flux de données et destinés aux architectures pouvant exploiter ce paradigme du calcul. Le but de cette thèse était constitué de plusieurs points relatifs à trois idées principales : premièrement, la présentation des architectures du calcul susceptibles d'être utilisées pour le calcul morphologique à bas prix ; deuxièmement, l'utilisation de l'approche formelle pour la description des algorithmes et troisièmement, le descriptif des algorithmes eux-mêmes et de leurs résultats expérimentaux.
De ce point de vue, les sujets que nous avons choisis de traiter dans cette thèse sont les suivants :

- Présenter les possibilités des architectures grand public pour le traitement morphologique en flux de données en exploitant leurs capacités de l'exécution parallèle SIMD sur les données paquetées à l'échelle des registres (on parle également des capacités SWAR) ; présenter les possibilités offertes par le jeu d'instructions multimédia de ces architectures et présenter les capacités du calcul parallèle à l'échelle des tâches et des fils d'exécution qui sont en train de devenir l'idée porteuse de construction des architectures grand public et pour lesquelles nous pouvons espérer de grands changements dans les prochaines années.
- Proposer une manière formelle pour la description généralisée des algorithmes liés à une architecture.
- Démontrer formellement, en définissant les skeletons algorithmiques, les principes de la construction des algorithmes pour la morphologie mathématique et exploiter le paradigme de traitement en flux en l'appliquant sur les opérations morphologiques couramment utilisées (dilatation / érosion, opérations géodésiques, fonction distance, nivellements).
- Construire les algorithmes morphologiques spécifiques pour les architectures multimédia avec les capacités SIMD.
- Explorer l'idée originale d'exécution par macro blocs pour la construction des algorithmes morphologiques, y compris l'idée de travail SIMD, l'idée des superpixels et des propagations SIMD dans l'image en employant la transposition directe sur un macro bloc pour assurer l'exécution dans les directions perpendiculaires à l'axe de stockage des données dans la mémoire.
- Démontrer les possibilités du calcul morphologique en flux de données sur les processeurs graphiques et proposer une description formelle de tels algorithmes pour ces processeurs.
- Effectuer des expériences sur diverses implémentations d'algorithmes de la morphologie mathématique afin d'évaluer les temps d'exécution et leur comportement sur des images de tailles variées.

Première partie de la thèse

Le travail que nous avons exposé a été divisé en deux parties. Dans la première, nous avons présenté les architectures cibles (chapitre 3) de cette thèse et les possibilités matérielles d'exécution qu'elles

offrent. Parmi les architectures qui peuvent être utilisées pour le calcul avec les flux de donnés, nous nous sommes intéressés en particulier aux processeurs du calcul général grand public (GPP) avec les capacités multimédia, surtout pour leur position actuelle sur le marché qui les rend très intéressants du point de vue applicatif. Malgré le fait que nous avons visé ce type d'architectures pour nos algorithmes et que nous avons testé ces algorithmes sur un processeur appartenant à ce type d'architectures, le fonctionnement de ces algorithmes n'est pas lié à une architecture du calcul parallèle ou séquentiel quelconque et leurs principes peuvent être réutilisés, notamment sur les architectures qui ont été nativement conçues pour le traitement des flux de données

Un autre groupe d'architectures que nous avons pu explorer et dont nous présentons la description dans cette partie est constitué des processeurs graphiques (GPU). Ces processeurs n'était pas destinés, dans leur fonction originale, au calcul de l'analyse d'image mais à leur synthèse. Vu les possibilités de programmation qu'ils offrent à nos jours et qui les rapprochent de plus en plus des applications générales qui ont besoin de la parallélisation massive (ce qui est le cas pour les algorithmes de la morphologie mathématique), nous avons exploré également la piste de leur possible utilisation pour les implémentations des opérations de base de la morphologie.

C'est également dans la première partie (chapitre 4) de cette thèse que nous avons introduit le formalisme fonctionnel, représenté par le Lambda calcul, en tant que moyen de spécification que nous avons adopté pour la description des algorithmes. Nous avons choisi comme outil de travail le langage fonctionnel Haskell implémentant la théorie du lambda calcul. Il s'est avéré particulièrement utile pour la description des procédés traitant les flux de données. Nous avons pu facilement exprimer le traitement des flux comme traitement des listes du Haskell et exprimer également les kernels d'exécution, propres au paradigme stream, en tant que fonctions appliquées aux éléments de ces listes.

Pour pouvoir exprimer le travail de la morphologie traitant le voisinage en termes de Lambda calcul, nous avons défini, pour formaliser le traitement sur les architectures GPP, un certain nombre de fonctions primitives, incluant les fonctions de passage d'un array à un stream de données, les fonctions d'échantillonnage des pixels pour pouvoir extraire les pixels désignés par l'élément structurant et les fonctions qui expriment le kernel d'exécution de l'opération morphologique à l'échelle d'un pixel. Nous avons utilisé la description des algorithmes généraux en tant que skeletons algorithmiques, définis pour le type polymorphe α. Ce fait s'est révélé utile quand nous sommes passé, à partir du traitement sur le voisinage des données scalaires, au traitement du voisinage des données paquetées contenant plusieurs éléments de base et exprimés en Haskell par un autre type, plus spécifique, qui décrivait les vecteurs paquetés – PVec I α.

Pour pouvoir effectuer le même travail sur les processeurs graphiques, nous avons construit un modèle formel du fonctionnement du pipeline graphique. La construction d'un algorithme qui est exécuté sur les GPU va se poursuivre comme la spécification des fonctions correspondant aux programmes maniant les divers blocs du pipeline graphique.

L'apport de ce formalisme a été particulièrement apprécié lors des vérifications de fonctionnement des descriptions présentées dans cette thèse car tous les algorithmes que nous décrivons dans le Lambda calcul sont également les fonctions du Haskell. Ainsi, nous avons pu faire appel au compilateur du Haskell et vérifier leur bonne syntaxe. De plus, il a été possible de tester, pour les algorithmes concrets, leur fonctionnement sur les données synthétiques en exécutant le programme du Haskell.

Deuxième partie de la thèse

La deuxième partie de cette thèse a été consacrée à la description des différentes façons de construire des algorithmes morphologiques pour les architectures orientées flux. Nous avons exploré, dans le chapitre 5, les principes qui entrent en jeu lors de la construction des algorithmes travaillant sur le voisinage, qui ne dépendent pas du sens du parcours et dont l'exécution peut être facilement parallélisée. Dans le même chapitre, nous avons présenté les algorithmes pour les processeurs graphiques. Les tests comparatifs pour les opérations de base de la morphologie mathématique qui mettaient en relation des implémen-

tations sur les GPU et diverses implémentations sur les GPP, y compris les algorithmes exploitant les capacités SIMD, ont révélé que pour les images de grande taille qui dépassent la capacité de la mémoire cache de notre architecture GPP, les processeurs graphiques montrent les meilleures performances.

Dans le chapitre suivant, le chapitre 6, nous avons présenté les algorithmes rapides pour les GPP qui exploitent les capacités SIMD de l'architecture multimédia et qui sont destinés au changement de l'axe de stockage des données paquetées. Nous avons présenté les algorithmes qui utilisent les fonctions shuffle, présentes dans tous les jeux d'instructions multimédia, qui effectuent l'opération choisie de changement de stockage dans $N\log_2 N$ applications des fonctions shuffle. Quatre algorithmes qui effectuent le changement souhaité et dont le principe de fonctionnement est très proche ont été présentés : la transposition par diagonale / anti-diagonale, la rotation de $+\frac{\pi}{2}$ et de $-\frac{\pi}{2}$. Nous avons souligné que dans les applications pratiques il s'agira le plus souvent de la transposition par diagonale qui assurera le changement de stockage des données paquetées.

Le chapitre 7 a été consacré aux algorithmes qui dépendent du sens de parcours prédéfini de l'image. Nous avons présenté la particularité de ces parcours lors du travail avec les données paquetées qui exigent une approche différente pour la construction de tels algorithmes que celle utilisée habituellement pour les éléments de base de l'image. Nous avons notamment exploré l'idée de quatre parcours directionnels lors du travail avec l'élément structurant de la forme du disque unité sur la grille carrée en 4-voisinage. Pour l'adapter au travail avec les vecteurs paquetés, nous faisons appel à la transposition par diagonale que nous utilisons comme moyen de changement de l'axe de stockage de données afin d'accéder aux données dans le sens perpendiculaire à celui de leur stockage dans la mémoire. Nous avons également exploré l'idée d'effectuer ce changement de stockage à l'échelle des macro blocs, évitant ainsi les opérations de transfert de données entre le processeur et la mémoire lors du stockage des résultats intermédiaires. Les exemples des opérations qui utilisent la structure des algorithmes ont été décrits dans ce chapitre. Nous avons présenté la fonction distance en tant qu'algorithme non-géodésique et dont le résultat est connu après une application de l'algorithme de propagation. Nous avons présenté également les nivellements, qui sont les filtres morphologiques et qui ont le caractère géodésique, que l'on emploie dans les applications de filtrage des images préservant les contours des objets.

Dans le chapitre 8 qui était consacré aux algorithmes de la dilatation / l'érosion par les éléments structurants de la forme des segments, nous avons exploré les stratégies de la parallélisation à l'échelle des données paquetées. Pour la construction d'un algorithme SIMD qui employait l'idée de l'algorithme de van Herk-Gil-Werman, nous avons réutilisé toutes les idées présentées dans les chapitres précédents dédiés aux algorithmes : la propagation de la valeur à l'intérieur des macro blocs, l'application de l'algorithme sur les macro blocs qui ne dépend pas d'un sens de parcours particulier et la transposition de l'image que nous avons utilisée pour changer l'axe de stockage des données lors de l'application de l'élément structurant qui nécessite l'accès aux données dans le sens perpendiculaire à celui de l'axe de stockage de données dans la mémoire.

Le chapitre 9 a été dédié à l'explication de la problématique de l'expression de la complexité d'un algorithme pour des fins pratiques. Nous y avons démontré que l'expression de la complexité relative au temps du calcul par le O, utilisé le plus souvent dans la littérature, n'est pas appropriée dans les conditions pratiques où nous estimons principalement le temps du calcul et où les coûts des accès à la mémoire et les coûts des opérations arithmétiques varient. Nous avons proposé d'exprimer la complexité d'une manière plus explicite qui mentionne ces différents coûts expressément et qui est exprimée par la fonction Θ. Dans ce même chapitre, nous montrons également que la problématique de l'estimation du temps du calcul pour les GPU n'est pas aussi directe car il s'agit d'une architecture distribuée. Nous discutons les temps des transferts des données entre les GPP et les GPU et l'influence de l'API.

Contribution de cette thèse

L'apport principal de cette thèse est la présentation d'un sujet important du point de vue pratique et l'exploration des capacités SIMD d'une architecture multimédia pour assurer l'exécution des algorithmes

morphologiques travaillant sur le voisinage peut raccourcir le temps du calcul. Nous avons exploré également l'utilisation d'un autre type d'architectures intéressant de ce point de vue – les processeurs graphiques – et nous avons démontré sur les résultats expérimentaux qu'ils sont, à présent, assez performants pour concurrencer les GPP afin d'assurer l'exécution des opérations morphologiques. L'apport secondaire de cette thèse est celui du formalisme fonctionnel que nous avons adopté pour la description de nos algorithmes travaillant sur les flux de données.

Perspectives et axes de recherche possibles

Les travaux effectués pendant la préparation de cette thèse ont évoqué certaines idées ou même des problématiques qui n'ont pas pu être explorées en entier ou qui n'ont pas trouvé de vraie place dans ce manuscrit. Nous les mentionnons dans la section des perspectives comme les sujets d'une possible prochaine exploration car nous trouvons qu'ils sont intéressants. La valorisation des algorithmes dans les architectures dédiées est également à envisager.

La première idée, qui est tout-à-fait intéressante et qui se joint aux idées mentionnées implicitement dans cette thèse, est l'utilisation des processeurs à plusieurs cœurs pour paralléliser le calcul morphologiques et les architectures à plusieurs fils d'exécution, logiques ou physiques. Étant donné que nos efforts d'expérimentation se sont concentrés sur les tests comparatifs effectués sur les processeurs avec un seul fil d'exécution, il serait envisageable, et même très important pour les applications pratiques, d'effectuer une étude comparative qui montrerait les gains de performance lors d'exécution des algorithmes décrits dans cette thèse sur les architectures parallèles qui peuvent exploiter le paradigme stream de traitement des données ; cela à l'échelle des tâches par le paradigme *Divise et conquiers* mais également à l'échelle des éléments des streams par la réplication fonctionnelle (où nous remplaçons la fonction `map` du Haskell par la fonction `farm`).

Une autre idée est relative aux implémentations des algorithmes sur les architectures spécialisées, telles que les architectures nativement construites pour l'exécution en stream (e.g. le processeur Imagine) ou les architectures dédiées à la morphologie mathématique (e.g. les architectures prototypées à l'aide des FPGA). Cette idée est à considérer autant de plus vu le déroulement des événements les plus récents où l'AMD vient d'annoncer[Pri06] l'introduction de la technologie ouvrant pour les FPGA un accès direct dans l'architecture des processeurs GPPMM. Cette technologie, nommée *Torrenza* et destinée pour la gamme des processeurs AMD Opteron, devrait profiter de la liaison rapide à la mémoire centrale et au processeur lui-même laissant ainsi un grand terrain libre pour les applications dédiées, y compris celles de la morphologie mathématique auxquelles nous nous intéressons le plus.

La construction des procédés du calcul en flux de données qui utilisent la propagation à base de règles et dépendant du contenu de l'image est également une idée à explorer. Cette problématique est souvent traitée par les files d'attente ou les files d'attente hiérarchiques et est également intéressante ; surtout si nous voulons explorer l'exécution en stream et profiter de la parallélisation par la réplication fonctionnelle. Notons que certains principes d'activation des pixels en parallèle qui sont utilisés lors du traitement des files d'attente sont décrits dans la thèse de Dominique Noguet[Nog98].

Pouvoir exécuter les algorithmes à base de files d'attente sur les processeurs graphiques est également un sujet important à étudier. Ce type de traitement est, à l'état actuel des technologies des GPU, difficile à effectuer car malgré le fait que les GPU peuvent sortir en même temps plusieurs valeurs correspondant à l'activation / non-activation des voisins d'un pixel, le placement des pixels à traiter au bon endroit de la file d'attente n'est pas trivial et demanderait plusieurs itérations de traitement du pipeline graphique. Ce qui se serait traduit par l'augmentation du temps du calcul nécessaire et ce qui, par conséquent, défavoriserait fortement les GPU dans un tel scénario d'utilisation. Les changements de la structure des processeurs graphiques nous semblent cruciaux pour pouvoir effectuer ce type de traitement afin d'obtenir des performances intéressantes.

Dans la partie introductive (cf. la section 3.4.6, page 55), nous avons mentionné Brook – l'outil de traitement des données en stream utilisant les GPU comme support du calcul. D'après nos expériences, il

semble inadapté, dans sa forme actuelle, au traitement des données par les méthodes de la morphologie mathématique. On pourrait envisager d'améliorer cet outil pour qu'il soit possible d'utiliser le type de nombres entiers de 8 bits, couramment utilisé dans le traitement d'images, et y incorporer les méthodes de traitement sur les GPU que nous avons présentées dans cette thèse.

L'approche formelle par les expressions du Lambda calcul que nous avons choisie pour la description de nos algorithmes pourrait être transposée à l'idée de la spécification formelle constructive afin de l'utiliser non seulement pour la description mais également pour l'autocréation d'un véritable programme en terme d'instructions pour une architecture quelconque. La théorie des types[Rys05] exploite cette idée et elle constitue un système qui est à la fois le système de la logique mathématique et du langage de programmation où l'action de prouver la validité d'un théorème est équivalente à la construction d'un programme à partir de sa spécification formelle. Les représentants des outils pour prouver ces théorèmes sont les *prouveurs des théorèmes* tels qu'Alf, Coq et autres, cf. l'article[Nog02] qui compare leurs capacités.

La dernière idée que nous évoquons comme un axe possible de recherche appliquée et qui sera, nous le croyons, sûrement exploré dans l'avenir proche consiste en l'étude de l'utilisation possible des consoles de jeux pour le calcul morphologique et, plus généralement, de l'analyse des images. En effet, les consoles de jeux combinent dans une seule architecture dédiée au calcul particulier des jeux vidéo une puissante unité (parallélisée) de calcul général et une unité puissante de calcul dédiée à la synthèse des images. Cette combinaison offre de belles possibilités d'appliquer toutes les idées décrites dans cette thèse, non seulement celles consacrées aux processeurs généraux, mais également celles relatives aux processeurs graphiques.

Cette page est blanche par intention

Annexe

Cette page est blanche par intention

Annexe A
Fonctions pour assurer l'exécution en cycles

Il est propre à la programmation impérative d'utiliser les cycles et de les gérer par les variables sauvegardant l'état pour exprimer l'itération actuelle. En revanche, l'approche fonctionnelle à la programmation ne travaille pas avec l'information sur l'état du programme et les cycles ne sont pas définis a priori. L'exécution itérative se poursuit par les appels récursifs de la fonction qui évalue elle-même si la récursion doit prendre fin ou si elle doit se poursuivre.

Haskell fournit les fonctions de base pour assurer l'exécution itérative mais le choix final de la manière exacte de cette exécution est à définir par l'utilisateur.

La fonction iterate, standard du Haskell assure l'application itérative de la fonction f sur x. Elle crée une liste infinie qui contient comme premier élément la variable d'entrée x, comme deuxième le résultat de la première itération, puis de la deuxième etc. :

```
iterate  ::  (α → α) → α → [α]
iterate  f x = x : iterate f (f x)
```

Par exemple, la fonction

```
iterate (+1) 1
```

a pour résultat la liste infinie $[1, 2, 3, 4, 5, ...]$. Pour obtenir un nombre défini d'itérations à partir de cette liste, nous utilisons la fonction take, standard du Haskell, qui retourne la liste finie composée de n premiers éléments de la liste $(x : xs)$:

```
take  ::  Int → [α] → [α]
take 0 _ = []
take _ []= []
take n (x:xs)
   | n > 0 = x : take (n−1) xs
take _ _ = error "take"
```

Pour obtenir le résultat de l'exécution itérative, il suffit de prendre le dernier élément de la liste créée préalablement par les fonctions **iterate** et **take**. La fonction last, standard du Haskell, assure cette possibilité :

```
last  ::  [α] → α
last [x] = x
last (_:xs) = last xs
```

Ainsi, l'exécution de n itérations de la fonction f sur les données x est assurée par l'expression suivante :

```
last ∘ (take (n+1)) ∘ ( iterate f ) $ x
```

Cette page est blanche par intention

Annexe B
Définitions des fonctions utilitaires en Haskell

Fonctions c3e *et* c4e *de création des vecteurs de couleur*

La fonction c3e crée à partir d'un triplet d'éléments de couleur CElmnt un vecteur de couleur composé C en utilisant la fonction pvec, cf. 4.3.1.3 page 64 :

```
c3e              ::  (CElmnt, CElmnt, CElmnt) → C
c3e (c₁,c₂,c₃) = pvec (1,3) [(1,c₁), (2,c₂), (3,c₃)]
```

Suivant la même logique, la fonction c4e crée à partir d'un quadruplet d'éléments de couleur CElmnt un vecteur de couleur composé C :

```
c4e                  ::  (CElmnt, CElmnt, CElmnt, CElmnt) → C
c4e (c₁,c₂,c₃,c₄) = array (1,4) [(1,c₁), (2,c₂), (3,c₃), (4,c₄)]
```

Fonctions p2D *et* p3D *de création des points*

La fonction p2D crée un point de 2D à partir d'un tuple de coordonnées :

```
p2D  ::  (Pos, Pos) → P
p2D (x₁,x₂) = pvec (1,2) [(1,x₁), (2,x₂)]
```

La fonction p3D crée un point de 3D à partir d'un triplet de coordonnées :

```
p3D  ::  (Pos, Pos, Pos) → P
p3D (x₁,x₂,x₃) = pvec (1,3) [(1,x₁), (2,x₂), (3,x₃)]
```

Fonctions mkTX, getArFromTX *et* getTXBFromTX *de manipulation des textures*

La fonction mkTX crée une texture à partir de ses composantes. La fonction (,) prend deux paramètres et crée un tuple.

```
mkTX          ::  Ar (I,I) C → [TXB] → TX
mkTX ar txbs =  (,) ar txbs
```

La fonction getArFromTX retourne l'array de couleur à partir de la texture TX :

```
getArFromTX  ::  TX → Ar (I,I) C
getArFromTX (x, _) = x
```

La fonction getTXBFromTX retourne la valeur de bord TXB d'une texture TX :

```
getTXBFromTX  ::  TX → TXB
getTXBFromTX (_, x) = x!!0
```

Fonction mkV *de création d'un vertex*

La fonction mkV crée un vertex à partir de ses composantes. La fonction (,,) prend trois paramètres et crée un triplet.

```
mkV              ::  P → [(CI,C)] → [(TXI, TXP)] → V
mkV p cs xps = (,,) p cs xps
```

Fonction mkF *de création d'un fragment*

La fonction mkF crée un fragment à partir de ses composantes. La fonction (,,,) prend quatre paramètres et crée un quadruplet.

```
mkF              ::  P → Dpth → [(CI,C)] → [(TXI, TXP)] → F
mkF p d cs xps = (,,,) p d cs xps
```

Fonction mkEnv *de création de l'environnement du pipeline graphique*

La fonction mkEnv crée, à partir d'un framebuffer FB et d'une liste des textures TX , l'environnement du travail Env du GPU :

```
mkEnv        ::  FB → [TX] → Env
mkEnv fb txs = (,) fb txs
```

Fonctions getTXs, getFB *de manipulation de l'environnement*

La fonction getTXs retourne la liste des textures [TX] à partir de l'environnement Env du pipeline graphique :

```
getTXs           ::  Env → [TX]
getTXs (_, txs) = txs
```

La fonction getFB retourne le framebuffer FB à partir de l'environnement Env :

```
getFB          ::  Env → FB
getFB (fb, _) = fb
```

Dimension des arrays, fonctions dimsAr1D *et* dimsAr2D

La fonction dimsAr1D retourne la taille d'un vecteur PVec ou d'un array 1D :

```
dimsAr1D    ::  Ar I α → I
dimsAr1D ar = q−p+1
  where (p,q) = bounds $ ar;
```

La fonction dimsAr2D retourne un tuple correspondant aux dimensions d'un array 2D dans la première et deuxième coordonnée, respectivement :

```
dimsAr2D    ::  Ar (I,I) α → (I,I)
dimsAr2D ar = (r−p+1, s−q+1)
  where ((p,q), (r,s)) = bounds $ ar;
```

Tests SIMD et déplacement conditionnel SIMD

Pour pouvoir assurer l'évaluation des expressions conditionnelles en SIMD, nous définissons la fonction de test testSIMD qui applique la fonction *cnd* du test logique entre chaque élément des deux vecteurs paquetés pv_1 et pv_2. Le résultat de cette fonction est un masque représenté par un vecteur paqueté dont les éléments sont les valeurs booléennes et qui contiennent soit la valeur True, soit la valeur False dans le cas où le test a réussi ou échoué, respectivement.

```
testSIMD :: PVec I α → (α → α → Bool) → PVec I α → PVec I Bool
testSIMD pv1 cnd pv2 = listArray (bounds $ pv1) (map2 cnd (elems$pv1) (elems$pv2))
```

La fonction de déplacement conditionnel cndmoveSIMD va utiliser un masque *msk* pour constituer un vecteur paqueté à partir de deux vecteurs paquetés d'entrée pv_1 et pv_2. Selon les valeurs du masque True ou False, les éléments du pv_1 ou pv_2, respectivement, sont copiés dans le vecteur résultant.

```
cndmoveSIMD :: PVec I Bool → PVec I α → PVec I α → PVec I α
cndmoveSIMD msk pv1 pv2 = listArray (bounds $ pv1) (map2 ( λ msk x y → if msk then x else y)
                                                         (elems$pv1)
                                                         (elems$pv2)
                                                   )
```

Fonction mapping travaillant avec plusieurs valeurs d'entrée

La fonction map2 effectue le mapping d'une fonction prenant deux arguments. Son fonctionnement est identique à celui de la fonction **zipWith** du Haskell :

```
map2 = zipWith
```

Cette page est blanche par intention

Liste des termes et des abréviations

1D	1 Dimension
2D	2 Dimensions
3D	3 Dimensions
AGP	Advanced Graphics Port
ALU	*Aritmetical and Logical Unit*, unité arithmétique et logique
API	*Application Programming Interface*, interface de programmation d'applications
ARB	*OpenGL Architectural Review Board*
ASIC	*Application Specific Integrated Circuit*, circuit intégré spécifique à l'application
blending	Le processus de combinaison de deux ou plusieurs choses afin de les mixer entièrement
CCD	*Charge Coupled Device*
CISC	*Complex Instruction Set Computer/Computing*, ordinateur/calcul à jeu d'instructions réduit
CMOS	*Complementary Metal Oxide Semi-conductor*
CMP	*Chip-level Multiprocessing*, multiprocessing au niveau de la puce
CMT	*Chip-level Multithreading*, multithreading au niveau de la puce
CPU	*Central Processing Unit*, unité centrale de calcul
DMIPS	*Dhrystone MIPS*, indicateur de performance dérivé des résultats du test Dhrystone [Wik06d]
FLOPS	*Floating-Point Operations Per Second*, q.v. [Wik06f]
FPGA	*Field Programmable Gate Array*, circuit intégré programmable
fps	*Frames per second*, trames par seconde
fragment	structure de données d'un GPU, contient des coordonnés 2D de la position sur l'écran, la coordonnée z et l'information sur la(les) couleur(s)
GFLOPS	*Giga Floating-Point Operations Per Second*, = 1024 MFLOPS = 1048576 FLOPS
GPGPU	*General Processing on Graphics Processing Units*
GPP	*General Purpose Processor*, processeur à usage général
GPPMM	*General Purpose Processor with MultiMedia extensions*, processeur à usage général avec les fonctionnalités multimédia
GPU	*Graphique Processing Unit*, unité du calcul graphique, processeur graphique
HAL	*Hardware Abstraction Layer*, couche d'abstraction du matériel
HDTV	*High Definition TeleVision*, télévision à haute définition
HGW	L'algorithme de *van Herk-Gil-Werman*
PC	*Personal Computer*, ordinateur personnel
ko	*Kilo-Octet*, 1 ko = 1024 octets = 1024*8 bits
LPE	*Ligne de Partage des Eaux*
nD	n-Dimensions
RISC	*Reduces Instruction Set Computer/Computing*, ordinateur/calcul à jeu d'instructions réduit
SIMD	*Single Instruction (stream), Multiple Data (stream)*
SISD	*Single Instruction (stream), Single Data (stream)*
SKIZ	*SKeleton by Influence Zones*, Squelette par zones d'influence, e.g. [Beu90]
SPMD	*Single Programme, Multiple Data (stream)*

SMT	*Simultaneous MultiThreading*, multithreading simultané
SWAR	*Single instruction multiple data Within A Register*
texel	*TEXture ELement*, structure de données d'un GPU, contient des information d'un élément de texture
MFLOPS	*Mega Floating-Point Operations Per Second*, = 1024 FLOPS
MIMD	*Multiple Instruction (stream), Multiple Data (stream)*
MIPS	*Million (integer) Instructions Per Second*, q.v. [Wik06h]
MISD	*Multiple Instruction (stream), Single Data (stream)*
Mo	*Méga-Octet*, 1 Mo = 1024 ko = 1048576 octets = 1048576*8 bits
MT	*MultiThreading*
vertex	structure de données d'un GPU, contient des coordonnées d'un sommet d'une forme géométrique et éventuellement d'autres informations (couleurs, position dans la texture, etc.)
VLIW	*Very Long Instruction Word*, architecture à mot d'instruction élargi
VMT	*Virtual MultiThreading*, multithreading virtuel
voxel	élément d'un volume discrétisé

Liste des figures

Cette page est blanche par intention

Liste des tableaux

Cette page est blanche par intention

Bibliographie

[AD03] Marco ALDINUCCI et Marco DANELUTTO : An Operational Semantics for Skeletons. Présentation. ParCo 2003, 2003. Disponible sur : http://www.di.unipi.it/~aldinuc/paper_files/2004_sem_parco03.pdf. [réf. du : 09 may 2006]. Format PDF. 26

[ADGkG96] Peter AU, John DARLINGTON, Moustafa M. GHANEM et Yi ke GUO : Co-ordinating heterogeneous parallel computation. volume 1, pages 601–614. Springer-Verlag, août 1996. Disponible sur : http://hpc.doc.ic.ac.uk/environments/coordination/papers/europar96.ps. [réf. du : 20 oct 2005]. Format PS. 26

[Ake03] Kurt AKELEY : Data Storage and Transfer in OpenGL. Présentation. SIGGRAPH 2003, juillet 2003. Disponible sur : http://developer.nvidia.com/docs/IO/8229/Data-Xfer-Store.pdf. [réf. du : 14 may 2006]. Format PDF. 183

[AMD06] AMD : Multi-Core Processors - the Next Evolution in Computing. 2006. Disponible sur : http://multicore.amd.com/WhitePapers/Multi-Core_Processors_WhitePaper.pdf. [réf. du : 04 apr 2006]. Format PDF. 2006. 49

[ARM06] ARM : ARM11 MPCore. 2006. Disponible sur : http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html. [réf. du : 03 apr 2006]. Format HTML. 2006. 43, 49

[ATI06] ATI : Radeon® X850 - The Performance Leader, Reviews. 2006. Disponible sur : http://www.ati.com/products/radeonx850/reviews.html. [réf. du : 06 feb 2006]. Format HTML. 2006. 43

[Bac77] John BACKUS : Can Programming Be Liberated from the von Neumann Style ? A Functional Style and Its Algebra of Programs. *ACM Turing Award Lecture*, 21(8):613–641, août 1977. Disponible sur : http://www.stanford.edu/class/cs242/readings/backus.pdf. [réf. du : 31 jul 2005]. Format PDF. 59

[Bag02] Daniele BAGNI : DSP and System-on-Chip. Présentation. DSP Application Day 2002, 2002. 37, 39, 42

[BB94] Henk P. BARENDREGT et Erik BARENDSEN : Introduction to Lambda Calculus. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1994. Disponible sur : http://citeseer.ist.psu.edu/barendregt94introduction.html. [réf. du : 11 oct 2005]. Format PS, PDF, PS.GZ. 59, 60, 61

[BBM+96] Andreas BIENIEK, Hans BURKHARDT, Heiko MARSCHNER, Gerald SCHREIBER et Michael NÖLLE : A Parallel Watershed Algoritm. Rapport technique, Technische Informatik I, TU-HH, Universität Freiburg, juin 1996. Disponible sur : ftp://www.ti1.tu-harburg.de/pub/papers/bie:etal:ib96.ps. [réf. du : 12 jul 2005]. Format PS. 28

[BD93] Andy D. BEN-DYKE : Architectural Taxonomy - A Brief Review. 1993. Disponible sur : http://phase.hpcc.jp/mirrors/parallel/faqs/comp-architecture-taxonomy. [réf. du : 11 jul 2005]. Format TXT. 1993. 31

[Beu84] Serge BEUCHER : Implantation d'un logiciel de Morphologie Mathématique sur calculateur Propal 2. Rapport technique N-923, Centre de Géostatistique et de Morphologie Mathématique, Ecole Nationale Supérieure des Mines de Paris, 35, rue Saint Honoré, 77300 Fontainebleau CEDEX, octobre 1984. Rapport final. 46

[Beu90] Serge BEUCHER : *Segmentation d'images et Morphologie mathématique*. Thèse de doctorat, Ecole Nationale Supérieure des Mines de Paris, juin 1990. Disponible sur : http://cmm.ensmp.fr/~beucher/publi/SB_these.pdf. [réf. du : 22 apr 2006]. Format PDF. 99, 205

[BFH+04a] Ian BUCK, Tim FOLEY, Daniel HORN, Jeremy SUGERMAN, Kayvon FATAHALIAN, Mike HOUSTON et Pat HANRAHAN : Brook for GPUs : Stream Computing on Graphics Hardware. *SIGGRAPH 2004*, 2004. Disponible sur : http://graphics.stanford.edu/papers/brookgpu/brookgpu.pdf. [réf. du : 02 aug 2005]. Format PDF. 56

[BFH+04b] Ian BUCK, Tim FOLEY, Daniel HORN, Jeremy SUGERMAN, Kayvon FATAHALIAN, Mike HOUSTON et Pat HANRAHAN : Brook for GPUs : Stream Computing on Graphics Hardware. Présentation. SIGGRAPH 2004, 2004. Disponible sur : http://graphics.stanford.edu/papers/brookgpu/buck.Brook.pdf. [réf. du : 02 aug 2005]. Format PDF. 56

[BHM+00] Prasenjit BISWAS, Atsushi HASEGAWA, Srinivas MANDAVILLE, Mark DEBBAGE, Andy STURGES, Fumio ARAKAWA, Yasuhiko SAITO et Kunio UCHIYAMA : SH-5 : The 64-bit SuperH Architecture. *IEEE Micro*, 20(4):28–39, juilletaoût 2000. Disponible sur : http://dx.doi.org/10.1109/40.865864. [réf. du : 17 mar 2006]. Format PDF. INSPEC Accession Number :6678832. 38

[Bil92] Michel BILODEAU : *Architecture logicielle pour processeur de morphologie mathématique*. Thèse de doctorat, Ecole Nationale Supérieure des Mines de Paris, Centre de Morphologie Mathématique, janvier 1992. 28

[BM83] Serge BEUCHER et F. MARTIN : Implantation d'un logiciel de Morphologie Mathématique sur calculateur Propal 2. Rapport technique N-803, Centre de Géostatistique et de Morphologie Mathématique, Ecole Nationale Supérieure des Mines de Paris, 35, rue Saint Honoré, 77300 Fontainebleau CEDEX, mars 1983. Rapport technique no. 1. 46

[BM98] Andreas BIENIEK et Alina MOGA : A Connected Component Approach to the Watershed Segmentation. *In Mathematical Morphology and its Applications to the Image and Signal Processing*, volume 12, pages 215–222. Kluwer Academic Publishers, 1998. Disponible sur : ftp://ftp.informatik.uni-freiburg.de/papers/lmb/bi:mo:ismm98.ps.gz. [réf. du : 5 mar 2005]. Format PS.GZ. 28

[Boh02] Mark BOHR : Intel's 90 nm Technology : Moore's Law and More. Présentation. Intel Developer Forum Fall 2002, septembre 2002. Disponible sur : ftp://download.intel.com/technology/silicon/Bohr_IDF_0902.pdf. [réf. du : 17 jul 2005]. Format PDF. 20

[Boh04] Mark BOHR : Intel's 65 nm Process Technology. Présentation. Intel Developer Forum, 2004. Disponible sur : ftp://download.intel.com/technology/silicon/IRDS002_65nm_logic_process_100_percent.pdf. [réf. du : 17 jul 2005]. Format PDF. 19, 20

[Bra02] Jaromir BRAMBOR : Implementation Notes On Binary Dilation And Erosion On 64bit SH-5 Processor (Principles of binary dilation and erosion using Minkowski Addition on 64-bit RISC SH-5 SuperH Multimedia Architecture). Rapport technique, Centre de Morphologie Mathematique, Ecole Nationale Superieure des Mines de Paris, Fontainebleau, October 2002. Disponible sur : http://cmm.ensmp.fr/~brambor/publications/(Bramb02)Implementation-Notes-On-Binary-Dilation-and-Erosion-On-64bit-SH5-Processor.en.pdf. [réf. du : 15 jul 2005]. Format PDF. 21, 65

[Bra05] Jaromir BRAMBOR : MorphoMedia documentation. Rapport technique, Centre de Morphologie Mathématique Ecole Nationale Supérieure des Mines de Paris, ARMINES, juin 2005. 41, 42, 65, 207

[Bre65] Jack E. BRESENHAM : Algorithm for Computer Control of a Digital Plotter. *IBM Systems Journal*, 4(1):25–30, 1965. Disponible sur : http://www.research.ibm.com/journal/sj/041/ibmsjIVRIC.pdf. [réf. du : 06 jan 2006]. Format PDF. Reprinted in Interactive Computer Graphics, Herbert Freeman ed., IEEE catalog no. EHO 156-0, Library of Congress no. 79-91237, 1980, and Seminal Graphics : Pioneering Efforts That Shaped The Field, Rosalee Wolfe ed., ACM SIGGRAPH, ACM order no. 435985, ISBN 1-58113-052-X, 1998. 167

[Bre05] Clay P. BRESHEARS : Intel Threading Tools and OpenMP. avril 2005. Disponible sur : http://cache-www.intel.com/cd/00/00/21/70/217017_217017.pdf. [réf. du : 31 mar 2006]. Format PDF. avril 2005. 50

[BW95] V. Michael Jr. BOVE et John A. WATLINGTON : Cheops : A Reconfigurable Data-Flow System for Video Processing. *IEEE Transactions on Circuits and Systems for Video Technology*, pages 140–149, avril 1995. Disponible sur : http://web.media.mit.edu/~wad/cheops_CSVT/cheops.pdf. [réf. du : 16 feb 2006]. Format PDF. 46

[Cam96] Duncan K. G. CAMPBELL : Towards the Classification of Algorithmic Skeletons. décembre 1996. Disponible sur : http://www.cs.uiuc.edu/homes/snir/PPP/skeleton/classification.pdf. [réf. du : 04 apr 2006]. Format PDF. décembre 1996. 27

[Cas03] S. CASS : Supercheap Supercomputer. *Spectrum, IEEE*, 40(7):17–17, juillet 2003. Disponible sur : http://dx.doi.org/10.1109/MSPEC.2003.1209605. [réf. du : 21 mar 2006]. Format PDF. 37

[CDPS03] Joao Luiz Dihl COMBA, Carlos A. DIETRICH, Christian A. PAGOT et Carlos E. SCHEIDEGGER : Computation on GPUs : From a Programmable Pipeline to an Efficient Stream Processor. 2003. Disponible sur : http://www.sci.utah.edu/~cscheid/pubs/rita_survey.pdf. [réf. du : 02 aug 2005]. Format PDF. 56

[CHD+02] Yen-Kuang CHEN, Matthew HOLLIMAN, Eric DEBES, Sergey ZHELTOV, Alexander KNYAZEV, Stanislav BRATANOV, Roman BELENOV et Ishmael SANTOS : Media Applications on Hyper-Threading Technology. *In Intel Technology Journal (2002, Volume 06 Issue 01)*[Int02], pages 47–57. Disponible sur : ftp://download.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf. [réf. du : 31 mar 2006]. Format PDF. 50

[Col89] Murray I. COLE : *Algorithmic Skeletons : Structured Management of Parallel Computation*. MIT Press, 1989. Disponible sur : http://homepages.inf.ed.ac.uk/mic/Pubs/skeletonbook.ps.gz. [réf. du : 11 oct 2005]. Format PS.GZ. 26

[Cou02] Rémi COUDARCHER : *Composition de squelettes algorithmiques : application au prototypage rapide d'applications de vision*. Thèse de doctorat, Laboratoire des Sciences et Matériaux pour l'Electronique, et d'Automatique (LASMEA), UNIVERSITE BLAISE PASCAL - CLERMONT-FERRAND II, décembre 2002. Disponible sur : http://tel.ccsd.cnrs.fr/documents/archives0/00/00/33/50/tel-00003350-02/tel-00003350.pdf. [réf. du : 20 oct 2005]. Format PDF. 26

[CT93] Michel COSNARD et Denis TRYSTRAM : *Algorithmes et architectures paralleles*. InterEditions, Paris, 1993. 31, 32

[Cui99] Olivier CUISENAIRE : *Distance Transformations : Fast Algorithms and Applications to Medical Image Processing*. Thèse de doctorat, Université Catholique de Louvain, octobre 1999. 28, 147

[CW02] Cem CEBENOYAN et Matthias WLOKA : Graphics Performance : Balancing the Rendering Pipeline. Présentation. GDC2002, 2002. Disponible sur : http://www.cs.virginia.edu/~gfx/Courses/2002/RealTime.fall.02/GDC2002_PipePerformance.ppt. [réf. du : 13 may 2006]. Format PPT. 183

[Dal03] Daniel Sanchez-Crespo DALMAU : *Core Techniques and Algorithms in Game Programming*. New Riders Publishing, novembre 2003. 55

[Dar98] John DARLINGTON : Generic Co-ordination Forms for Parallel Application Construction. Rapport technique, Imperial College, 1998. Disponible sur : http://hpc.doc.ic.ac.uk/environments/coordination/final.ps. [réf. du : 20 oct 2005]. Format PS. 26

[DB05] Marc Van DROOGENBROECK et M. BUCKLEY : Morphological erosions and openings : Fast algorithms based on anchors. *Journal of Mathematical Imaging and Vision, Special Issue on Mathematical Morphology after 40 Years*, 22(2-3):121–142, mai 2005. Disponible sur : http://dx.doi.org/10.1007/s10851-005-4886-2. [réf. du : 09 may 2005]. Format HTML. Springer Netherlands. 168, 174

[DD03] Eva DEJNOZKOVA et Petr DOKLADAL : A multiprocessor architecture for PDE-based applications. *Visual Information Engeneering*, -:–, juillet 2003. Disponible sur : http://cmm.ensmp.fr/~dejnozke/professionnel/Dejnozkova_VIE2003_final.pdf. [réf. du : 17 jul 2005]. Format PDF. 28

[DD04] Eva DEJNOZKOVA et Petr DOKLADAL : Asynchonous Multi-core Architecture for Level Set Methods. *IEEE International Conference on Accoustics, Speech and Signal Processing ICASSP*, -:–, mai 2004. Disponible sur : http://cmm.ensmp.fr/~dejnozke/professionnel/dejnozkova.pdf. [réf. du : 17 jul 2005]. Format PDF. 28

[DD06] Renaud DARDENNE et Marc Van DROOGENBROECK : The libmorpho library. 2006. Disponible sur : http://www.ulg.ac.be/telecom/research/libmorpho.html. [réf. du : 03 july 2006]. Format HTML. 2006. 168, 174

[Dej04] Eva DEJNOZKOVA : *Architecture dédié au traitement d'images basé sur les equations aux dérivées partielles*. Thèse de doctorat, Ecole des Minbes de Paris, mars 2004. Disponible sur : http://cmm.ensmp.fr/~dejnozke/these_print_dejnozkova_v2.pdf. [réf. du : 17 jul 2005]. Format PDF. 24, 28

[DFH+93] J. DARLINGTON, A. J. FIELD, P. G. HARRISON, P. H. J. KELLY, D. W. N. SHARP, Q. WU et R. L. WHILE : Parallel Programming Using Skeleton Functions. *In* A. BODE, M. REEVE et G. WOLF, éditeurs : *PARLE '93 : Parallel Architectures and Languages Europe*, pages 146–160. Springer-Verlag, Berlin, DE, 1993. Disponible sur : http://citeseer.ist.psu.edu/darlington93parallel.html. [réf. du : 10 oct 2005]. Format PDF, PS. 26, 66, 67

[DGG+96] John DARLINGTON, Yike GUO, Moustafa GHANEM, Jin YANG, Kwok Tat Peter AU et Rami SIK : Coordinating Combined Parallel Vector and Scalar Computation. Rapport technique IFPC-TR-96-1, Imperial College, janvier 1996. Disponible sur : http://hpc.doc.ic.ac.uk/environments/coordination/papers/ifpc-tr-96-1.ps. [réf. du : 20 oct 2005]. Format PS. 26

[DGGT97] John DARLINGTON, Moustafa M. GHANEM, Yike GUO et H. W. TO : Performance models for coordinating parallel data classification. septembre 1997. Disponible sur : http://hpc.doc.ic.ac.uk/environments/coordination/papers/pcw97-datamining.ps. [réf. du : 20 oct 2005]. Format PS. 26

[DGTY95a] J. DARLINGTON, Y. GUO, H. W. TO et J. YANG : Parallel skeletons for structured composition. *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 19–28, juillet 1995. Disponible sur : http://hpc.doc.ic.ac.uk/environments/coordination/papers/ppopp.ps. [réf. du : 20 oct 2005]. Format PS. 26

[DGTY95b] John DARLINGTON, Yike GUO, Hing Wing TO et Jin YANG : Functional Skeletons for Parallel Coordination. *In Euro-Par '95 : Proceedings of the First International Euro-Par Conference on Parallel Processing*, volume 996, pages 55–66, London, UK, 1995. Springer-Verlag. Disponible sur : http://hpc.doc.ic.ac.uk/environments/coordination/papers/europar95.ps. [réf. du : 04 apr 2006]. Format PS. 26, 67

[DkGT95] John DARLINGTON, Yi ke GUO et Hing Wing TO : Stuctured Parallel Programming : Theory meets Practice. 1995. Disponible sur : http://hpc.doc.ic.ac.uk/environments/coordination/papers/book.ps. [réf. du : 18 oct 2005]. Format PS. 26

[DkGTY96] John DARLINGTON, Yi ke GUO, Hing Wing TO et Jin YANG : SPF : Structured Parallel Fortran. page 6, 1996. Disponible sur : http://hpc.doc.ic.ac.uk/environments/coordination/papers/pcw96-spf.ps. [réf. du : 15 dec 2005]. Format PS. 26

[DMO+92] Marco DANELUTTO, Robert Di MEGLIO, Salvatore ORLANDO, Susanna PELAGATTI et Marco VANNESCHI : A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 8(1-3):205–220, juillet 1992. Disponible sur : ftp://ftp.di.unipi.it/pub/project/p3l/florence.ps.gz. Format PS.GZ. pre-print. 26

[DNG00] John DARLINGTON, Steven NEWHOUSE et Yike GUO : Parallel Problem Solving Course 2000. Présentation. 2000. Disponible sur : http://hpc.doc.ic.ac.uk/PPS/PPS00ALL.ppt. [réf. du : 15 dec 2005]. Format PPT. 26

[DPRS01] Udo DIEWALD, Tobias PREUSSER, Martin RUMPF et Robert STRZODKA : Diffusion Models and Their Accelerated Solution in Image and Surface Processing. *Acta Mathematica Universitatis Comenianae*, 70(1): 15–31, 2001. 28

[DT95] J. DARLINGTON et H. W. TO : Building Parallel Applications Without Programming. *In* J. R. DAVY et P. M. DEW, éditeurs : *Abstract Machine Models for Highly Parallel Computers*, pages 140–154. Oxford University Press, 1995. Disponible sur : http://hpc.doc.ic.ac.uk/ala/papers/H.W.To/leeds.ps.Z. [réf. du : 12 oct 2005]. Format PS.Z. 26

[DT96] Marc Van DROOGENBROECK et Hugues TALBOT : Fast computation of morphological operations with arbitrary structuring elements. *Pattern Recogn. Lett.*, 17(14):1451–1460, 1996. Disponible sur : http://www.ulg.ac.be/telecom/publi/publications/mvd/anydila.pdf. 168

[Dun90] Ralph DUNCAN : A Survey of Parallel Computer Architectures. *IEEE Computer*, 23(2):5–16, février 1990. Disponible sur : http://dx.doi.org/10.1109/2.44900. [réf. du : 12 jul 2005]. Format PPT. 31, 33, 35

[Eng78] J. N. ENGLAND : A system for interactive modeling of physical curved surface objects. *In SIGGRAPH '78 : Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 336–340, New York, NY, USA, 1978. ACM Press. Disponible sur : http://doi.acm.org/10.1145/800248.807412. [réf. du : 16 feb 2006]. Format PDF. 56

[FBF00] Paolo FARABOSCHI, Geoffrey BROWN et Joseph A. FISHE : Lx : A Technology Platform for Customizable VLIW Embedded Processing. *In The 27th Annual International Symposium on Computer architecture 2000*, pages 203–213, New York, NY, USA, 2000. ACM Press. Disponible sur : http://citeseer.ist.psu.edu/faraboschi00lx.html. [réf. du : 20 mar 2006]. Format PDF. 37, 42

[FF88] Alain FOURNIER et Donald FUSSELL : On the power of the frame buffer. *ACM Trans. Graph.*, 7(2):103–128, 1988. Disponible sur : http://doi.acm.org/10.1145/42458.42460. [réf. du : 16 feb 2006]. Format PDF. 56

[FH05] M. J. FLYNN et P. HUNG : Microprocessor design issues : thoughts on the road ahead. *Micro, IEEE*, 25(3):16–31, 2005. Disponible sur : http://dx.doi.org/10.1109/MM.2005.56. [réf. du : 31 mar 2006]. INSPEC Accession Number :8512670. 23, 48

[FK03] Randima FERNANDO et Mark J. KILGARD : *The Cg Tutorial*. Addison-Wesley, février 2003. NVidia. 51, 55

[Fly66] Michael J. FLYNN : Very High-speed Computing Systems. *Proceedings of IEEE*, 54(12):1901–1909, décembre 1966. Disponible sur : http://ieeexplore.ieee.org/iel5/5/31091/01447203.pdf?tp=&arnumber=1447203&isnumber=31091. [réf. du : 12 jul 2005]. Format PPT. 31

[FS02] Inc. FREESCALE SEMICONDUCTOR : *MPC750 and MPC740 Microprocessors, Fact Sheet*, 2002. Disponible sur : http://www.freescale.com/files/32bit/doc/fact_sheet/MPC750FACT.pdf. [réf. du : 03 apr 2006]. Format PDF. 43

[GAA97] David Z. GEVORKIAN, Jaakko T. ASTOLA et Samvel M. ATOURIAN : Improving Gil-Werman Algorithm for Running Min and Max Filters. *In IEEE Trans. Pattern Anal. Mach. Intell.* [GW93], pages 526–529. Disponible sur : http://dx.doi.org/10.1109/34.589214. [réf. du : 06 jan 2006]. Format PDF. 167, 217

[Gav05a] Ilya GAVRICHENKOV : First Look at Presler : Intel Pentium Extreme Edition 955 CPU Review. décembre 2005. Disponible sur : http://www.xbitlabs.com/articles/cpu/display/presler.html. [réf. du : 14 mar 2006]. Format HTML. décembre 2005. 43

[Gav05b] Ilya GAVRICHENKOV : Intel Pentium 4 6XX and Intel Pentium 4 Extreme Edition 3.73GHz CPU Review. février 2005. Disponible sur : http://www.xbitlabs.com/articles/cpu/display/pentium4-6xx_8.html. [réf. du : 14 mar 2006]. Format HTML. février 2005. 43

[Gav06] Ilya GAVRICHENKOV : AMD Athlon 64 FX-60 CPU Review. janvier 2006. Disponible sur : http://www.xbitlabs.com/articles/cpu/display/athlon64-fx60.html. [réf. du : 14 mar 2006]. Format HTML. janvier 2006. 43

[GCRW+99] Antonio GENTILE, José L. CRUZ-RIVERA, D. Scott WILLS, Leugim BUSTELO, José J. FIGUEROA, Javier E. FONSECA-CAMACHO, Wilfredo E. LUGO-BEAUCHAMP, Ricardo OLIVIERI, Marlyn QUIÑONES-CERPA, Alexis H. RIVERA-RÍOS, Iomar VARGAS-GONZÁLES et Michelle VIERA-VERA : Real-Time Image Processing on a Focal Plane SIMD Array. *IPPS/SPDP Workshops*, pages 400–405, 1999. Disponible sur : http://ipdps.cc.gatech.edu/1999/wpdrts/gentile.pdf. [réf. du : 17 oct 2005]. Format PS, PS.GZ, PDF. 35

[Gha99] Moustafa Mahmoud Hafez GHANEM : *Structured Parallel Programming Using Performance Models and Skeletons*. Thèse de doctorat, University of London, Imperial College of Sciences, Technology and Medicine, Department of Computing, février 1999. Disponible sur : http://hpc.doc.ic.ac.uk/environments/coordination/papers/mmg.ps. [réf. du : 18 oct 2005]. Format ps. 26

[GHC] The GHCTEAM : *The Glorious Glasgow Haskell, Compilation System User's Guide, Version 6.4.1*. Disponible sur : http://www.haskell.org/ghc/docs/latest/users_guide.pdf. [réf. du : 01 dec 2005]. Format PDF. 62

[Gib94] Jeremy GIBBONS : An Introduction to the Bird-Meertens Formalism. *New Zealand Formal Program Development Colloquium Seminar*, page 12, 1994. Disponible sur : http://cs.anu.edu.au/people/Clem.Baker-Finch/AFP/nzfpdc-squiggol.ps.gz. 27

[GK02] Joseph (Yossi) GIL et Ron KIMMEL : Efficient Dilation, Erosion, Opening, and Closing Algorithms. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(12):1606–1617, 2002. Disponible sur : http://dx.doi.org/10.1109/TPAMI.2002.1114852. [réf. du : 06 jan 2006]. Format PDF. 167

[Gom01] Cristina GOMILA : *Mise en correspondance de partitions en vue du suivi d'objets*. Thèse de doctorat en morphologie mathématique, Ecole Nationale Supérieure des Mines de Paris, 2001. Disponible sur : http://cmm.ensmp.fr/~gomila/These_Gomila.pdf. [réf. du : 27 mar 2006]. Format PDF. 27, 36

[GPG] GPGPU. Disponible sur : http://www.gpgpu.org/. [réf. du : 09 mar 2006]. Format HTML. 22, 56

[Gra03] Kris GRAY : *Microsoft DirectX 9 Programmable Graphics Pipeline*. Microsoft Press, juillet 2003. CDROM. 54, 55

[Gre05] Will GREENWALD : Xbox 360 : BEWARE, I HUNGER ! ! ! décembre 2005. Disponible sur : http://reviews.cnet.com/4531-10921_7-6398157.html. [réf. du : 14 mar 2006]. Format HTML. décembre 2005. 43

[Gro02] Andy GROVE : Changing Vectors of Moore's Law. Présentation. International Electron Devices Meeting, dec 2002. Disponible sur : http://www.intel.com/pressroom/archive/speeches/grove_20021210.pdf. [réf. du : 07 mar 2006]. Format PDF. Intel Corporation. 19

[GS03] Charles W. GWYN et Peter J. SILVERMAN : EUVL : transition from research to commercialization. *Photomask and Next-Generation Lithography Mask Technology X*, 5130(1):990–1004, 2003. Disponible sur : http://dx.doi.org/10.1117/12.504239. [réf. du : 8 mars 2006]. Format PDF. 20

[GS05] John GOODACRE et Andrew N. SLOSS : Parallelism and the ARM Instruction Set Architecture. *IEEE Computer*, 38(7):42–50, juillet 2005. Disponible sur : http://dx.doi.org/10.1109/MC.2005.239. [réf. du : 04 apr 2006]. Format PDF. INSPEC Accession Number :8513365. 43

[GW93] J. GIL et M. WERMAN : Computing 2-D Min, Median, and Max Filters. *In IEEE Trans. Pattern Anal. Mach. Intell.* [GAA97], pages 504–507. Disponible sur : http://dx.doi.org/10.1109/34.211471. [réf. du : 06 jan 2006]. Format PDF. 167, 216

[GWH05] Nolan GOODNIGHT, Rui WANG et Greg HUMPHREYS : Computation on Programmable Graphics Hardware. *IEEE Comput. Graph. Appl.*, 25(5):12–15, 2005. Disponible sur : http://dx.doi.org/10.1109/MCG.2005.101. [réf. du : 30 nov 2005]. Format PDF. 56

[HAL04] Kevin HAWKINS, Dave ASTLE et André LAMOTHE : *OpenGL Game Programming*. The Premier Press, 2004. CDROM. 55

[HCSL02] Mark J. HARRIS, Greg COOMBE, Thorsten SCHEUERMANN et Anselmo LASTRA : Physically-Based Visual Simulation on Graphics Hardware. *Graphics Hardware*, pages 1–10, 2002. Disponible sur : http://www.markmark.net/cml/dl/HWW02_Harris_electronic.pdf. 56

[HE00] Matthias HOPF et Thomas ERTL : Accelerating Morphological Analysis with Graphics Hardware. 2000. Disponible sur : http://wwwvis.informatik.uni-stuttgart.de/eng/research/pub/pub2000/vmv00-hopf.pdf. [réf. du : 12 jul 2005]. Format PDF. 2000. 28, 118

[Hei95] Henk J.A.M. HEIJMANS : Mathematical Morphology : Basic Principles. 1995. Disponible sur : http://citeseer.ist.psu.edu/heijmans95mathematical.html. [réf. du : 07 mai 2006]. Format PS, PDF. 84

[HMJH05] Tim HARRIS, Simon MARLOW, Simon Peyton JONES et Maurice HERLIHY : Composable Memory Transactions. *ACM Conference on Principles and Practice of Parallel Programming 2005 (PPoPP'05)*, page 13, juin 2005. Disponible sur : http://research.microsoft.com/Users/simonpj/papers/stm/stm.pdf. [réf. du : 29 mar 2006]. 60

[HPF99] Paul HUDAK, John PETERSON et Joseph H. FASEL : A Gentle Introduction to Haskell 98. page 64, octobre 1999. Disponible sur : http://www.haskell.org/tutorial/haskell-98-tutorial.pdf. [réf. du : 9 apr 2006]. Format HTML. 61

[Ian88] Robert A. IANNUCCI : *A dataflow/von Neumann hybrid architecture*. Thèse de doctorat, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 1988. Disponible sur : http://hdl.handle.net/1721.1/14778. [réf. du : 39 mar 2006]. Format PDF. 47

[Int02] INTEL, éditeur. *Intel Technology Journal (2002, Volume 06 Issue 01)*, volume 06, février 2002. Disponible sur : ftp://download.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf. [réf. du : 29 mar 2006]. Format PDF. 214, 219, 221

[Int05] INTEL : Platform 2015 : Intel Processor and Platform Evolution for the Next Decade. page 12, 2005. Disponible sur : ftp://download.intel.com/technology/computing/archinnov/platform2015/download/Platform_2015.pdf. [réf. du : 17 jul 2005]. Format PDF. 20

[Int06a] INTEL : Intel First to Demonstrate Working 45nm Chips. 2006. Disponible sur : http://www.intel.com/technology/silicon/new_45nm_silicon.htm. [réf. du : 2 fev 2006]. Format HTML. 2006. 20

[Int06b] INTEL : Intel Integrated Perfromance Primitives. 2006. Disponible sur : http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm. [réf. du : 26 mar 2006]. Format HTML. 2006. 41

[Int06c] INTEL : Intel Pentium Processor Extreme Edition 995, Product Brief. 2006. Disponible sur : http://www.intel.com/products/processor/pentiumXE/index.htm. [réf. du : 14 mar 2006]. Format HTML. 2006. 43

[Int06d] INTEL : Processeur Intel Core Duo, Product Brief. 2006. Disponible sur : http://www.intel.com/products/processor/coreduo/product-brief.pdf. [réf. du : 04 apr 2006]. Format PDF. 2006. 49

[ITR] International Technology Roadmap for Semiconductors. Disponible sur : http://public.itrs.net/. [réf. du : 13 mars 2006]. Format HTML. 24

[Jon03] Simon Peyton JONES : *Haskell 98 Language and Libraries The Revised Report*, janvier 2003. Disponible sur : http://www.haskell.org/definition/haskell98-report.pdf. 60, 64

[Jou91] Guido K. JOURET : Compiling Functional Languages for SIMD Architectures. *Parallel and Distributed Processing, Proceedings of the Third IEEE Symposium on*, pages 79–86, décembre 1991. Disponible sur : http://dx.doi.org/10.1109/SPDP.1991.218294. [réf. du : 11 oct 2005]. Format PDF. ISBN : 0-8186-2310-1. 26

[KBR03] John KESSENICH, Dave BALDWIN et Randi ROST : *The OpenGL Shading Language version 1.051*, feb 2003. 55

[KDK+01] B. KHAILANY, W. DALLY, U. KAPASI, P. MATTSON, J. NAMKOONG, J. OWENS, B. TOWLES, A. CHANG et S. RIXNER : Imagine : Media Processing with Streams. *IEEE Micro*, 21(2):35–46, mars 2001. Disponible sur : http://cva.stanford.edu/cs99s/papers/imagine-ieeemicro.pdf. 37, 46

[KDR+02] Ujval J. KAPASI, William J. DALLY, Scott RIXNER, John D. OWENS et Brucek KHAILANY : The Imagine Stream Processor. *IEEE International Conference on Computer Design*, pages 282–288, 2002. Disponible sur : http://www.iccd-conference.org/proceedings/2002/17000282.pdf. 37, 46

[KEH+02] Steffen KLUPSCH, Markus ERNST, Sorin A. HUSS, Martin RUMPF et Robert STRZODKA : Real Time Image Processing based on Reconfigurable Hardware Acceleration. 2002. Disponible sur : http://numod.ins.uni-bonn.de/research/papers/public/KlErHuRuSt02.pdf. [réf. du : 22 apr 2006]. Format PDF. 28

[Ker97] Renaud KERIVEN : *Equation aux Dérivées Partielles, Evolution de Courbes et de Surfaces et Espaces d'Echelle : Application à la Vision par Ordinateur*. Thèse de doctorat, Ecole Nationale de Ponts et Chaussées, 1997. Disponible sur : http://tel.ccsd.cnrs.fr/tel-00005617/en/. [réf. du : 22 apr 2006]. Format PDF,PS. 28

[KKI04] Takashi KOMURO, Shingo KAGAMI et Masatoshi ISHIKAWA : A Dynamically Reconfigurable SIMD Processor for a Vision Chip. *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, 39(1):265–268, janvier 2004. Disponible sur : http://dx.doi.org/10.1109/JSSC.2003.820876. [réf. du : 10 jan 2006]. Format PDF. 65

[Kra04] Tom KRAZIT : Transmeta's M8800 Efficeon Processor Powers Down, New version of the processor uses only 3 watts of power at 1-GHz speed. octobre 2004. Disponible sur : http://www.pcworld.com/news/article/0,aid,118063,00.asp. [réf. du : 20 mar 2006]. Format HTML. octobre 2004. 37, 43, 44

[KRD+03] Ujval J. KAPASI, Scott RIXNER, William J. DALLY, Brucek KHAILANY, Jung Ho AHN, Peter MATTSON et John D. OWENS : Programmable Stream Processors. *IEEE Computer*, 36(8):54–62, août 2003. Disponible sur : ftp://cva.stanford.edu/pub/publications/ieeecomputer_stream.pdf. [réf. du : 07 nov 2005]. Format PDF. 46

[Kre05] Tino KREISS : Two's Company, Four's a WOW ! Sneak Preview of NVIDIA Quad GPU Graphics. 2005. Disponible sur : http://www.tomshardware.com/2005/12/14/sneak_preview_of_the_nvidia_quad_gpu_setup/print.html. [réf. du : 14 mar 2006]. Format HTML. 2005. 43, 51

[Lab97] Bell LABS : Bell Labs celebrates 50 years of the Transistor. 1997. Disponible sur : http://www.lucent.com/minds/transistor/pdf/inventor.pdf. [réf. du : 2 fev 2006]. Format PDF. 1997. 23

[LBB98] Alina LINDNER, Andreas BIENIEK et Hans BURKHARDT : PISA – Parallel Image Segmentation Algorithms. —, 1998. 1998. 28

[Lee00] Ruby B. LEE : Subword Permutation Instructions for Two-Dimensional Multimedia Processing in MicroSIMD Architectures. *Application-Specific Systems, Architectures, and Processors, 2000. Proceedings. IEEE International Conference on*, pages 3–14, juillet 2000. Disponible sur : http://dx.doi.org/10.1109/ASAP.2000.862373. [réf. du : 8 mar 2006]. Format PDF. INSPEC Accession Number : 6741658. 145

[Lem96] Fabrice LEMONNIER : *Architecture électronique dédiée aux algorithmes rapides de segmentation basés sur la morphologie mathématique*. Thèse de doctorat, Ecole Nationale Supérieure des Mines de Paris, décembre 1996. 28, 46

[LFB01] Ruby B. LEE, A. Murat FISKIRAN et Abdulla BUBSHAIT : Multimedia Instructions in IA-64. *IEEE International Conference on Multimedia and Expo 2001*, pages 281–284, août 2001. Disponible sur : http://dx.doi.org/10.1109/ICME.2001.1237694. [réf. du : 8 mar 2006]. Format PDF. 21, 145

[LM99] Lacezar LICEV et David MORKES : *Procesory - Architektura, funkce, pouziti*. Computer Press, 1999. 38

[Lov05] Anthony LOVESEY : A Comparison of Real Time Graphical Shading Languages. mars 2005. Disponible sur : http://www.cs.unb.ca/undergrad/html/documents/Lovesey_Senior_TechReport.pdf. Faculty of Computer Science, University of New Brunswick, Canada, mars 2005. 55

[Lue04] David LUEBKE : General-Purpose Computation on Graphics Hardware. 2004. Disponible sur : http://www.gpgpu.org/s2004/slides/luebke.Introduction.ppt. [réf. du : 12 jul 2005]. Format PPT. 2004. 23

[Mah05] Joseph M. MAHONEY : 3D Graphics Then and Now : From the CPU to the GPU. *Proceedings of the 5thWinona Computer Science Undergraduate*, pages 14–20, avril 2005. Disponible sur : http://cs.winona.edu/CSConference/2005proceedings/joe.pdf. 23

[MBH+02] Deborah T. MARR, Frank BINNS, David L. HILL, Glenn HINTON, David A. KOUFATY, J. Alan MILLER et Michael UPTON : Hyper-Threading Technology Architecture and Microarchitecture. *In Intel Technology Journal (2002, Volume 06 Issue 01)* [Int02], pages 4–15. Disponible sur : ftp://download.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf. [réf. du : 31 mar 2006]. Format PDF. 48

[Mei04] Arnold MEIJSTER : *Efficient Sequential and Parallel Algorithms for Morphological Image Processing*. Thèse de doctorat, Rijksuniversiteit Groningen, mars 2004. Disponible sur : http://rc60.service.rug.nl/~arnold/articles/DigThesis.pdf. [réf. du : 12 jul 2005]. Format PDF. ISBN 90-367-1978-x (digital version). 28

[Mey03] Fernand MEYER : Floodings, razings and levellings. Présentation. Centre de Morphologie Mathématique, mars 2003. 154

[MGR+05a] Victor MOYA, Carlos GONZALEZ, Jordi ROCA, Agustin FERNANDEZ et Roger ESPASA : Shader Performance Analysis on a Modern GPU Architecture. *Micro38 Barcelona, Spain*, pages 1–10, novembre 2005. Disponible sur : http://personals.ac.upc.edu/vmoya/docs/vmoya-ShaderPerformance.pdf. [réf. du : 2 fev 2006]. Format PDF. 54

[MGR+05b] Victor MOYA, Carlos GONZÁLEZ, Jordi ROCA, Agustín FERNÁNDEZ et Roger ESPASA : A Single (Unified) Shader GPU Microarchitecture for Embedded Systems. *HiPEAC 2005, 2005 International Conference on High Performance Embedded Architectures & Compilers*, pages 1–15, novembre 2005. Disponible sur : http://personals.ac.upc.edu/vmoya/docs/EmbeddedGPU.pdf. [réf. du : 2 fev 2006]. Format PDF. 54

[Mig04] Pascal MIGNOT : Pipeline graphique, Leçon n°2 : concept et organisation du pipeline graphique de DirectX 9. Présentation. 2004. Disponible sur : http://helios.univ-reims.fr/UFR/Info/Image/DirectX/cours/02-Pipelinegraphique.pdf. [réf. du : 10 apr 2006]. Format PDF. Université de Reims. 55

[Moo65] Gordon E. MOORE : Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, avril 1965. Disponible sur : ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf. [réf. du : 13 jul 2005]. Format PDF. 19

[Moo98] Gordon E. MOORE : Cramming More Components onto Integrated Circuits. *In* IEEE, éditeur : *Proceedings of the IEEE*, volume 86, pages 82–85, janvier 1998. Disponible sur : http://dx.doi.org/10.1109/JPROC.1998.658762. [réf. du : 13 jul 2005]. Format PDF. 19

[Moo03a] Gordon MOORE : No exponential is forever : but "Forever" can be delayed ! *In Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International* [Moo03b], pages 20–23. Disponible sur : http://dx.doi.org/10.1109/ISSCC.2003.1234194. [réf. du : 8 mars 2006]. Format PDF. INSPEC Accession Number : 7785691. 23, 219

[Moo03b] Gordon E. MOORE : No exponential is forever. Présentation. Solid-State Circuits Conference 2003, IEEE International, *In Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International* [Moo03a], pages 20–23. Disponible sur : http://ieeexplore.ieee.org/xpl/multimedia.jsp?arnumber=1234194. [réf. du : 15 avr 2006]. Format ZIP. INSPEC Accession Number : 7785691. 23, 219

[MR96] A. MEIJSTER et J. B. T. M. ROERDINK : Computation of Watersheds Based on Parallel Graph Algorithms. *Mathematical Morphology and its Applications to Image and Signal Processing*, pages 305–312, 1996. 28

[NH00] Desikachari NADADUR et Robert M. HARALICK : Recursive binary dilation and erosion using digital line structuring elements in arbitrary orientations. *In Proc. SPIE Vol. 3026, p. 95-105, Nonlinear Image Processing VIII, Edward R. Dougherty ; Jaakko T. Astola ; Eds.* [NHS97], pages 749–759. Disponible sur : http://dx.doi.org/10.1109/83.841511. [réf. du : 06 jan 2006]. Format PDF. 167, 220

[NHS97] D. NADADUR, R. M. HARALICK et F. H. SHEEHAN : Recursive binary dilation using digital line-structuring elements in arbitrary orientations. *In Proc. SPIE Vol. 3026, p. 95-105, Nonlinear Image Processing VIII, Edward R. Dougherty ; Jaakko T. Astola ; Eds.* [NH00], pages 95–105. Disponible sur : http://adsabs.harvard.edu/cgi-bin/nph-bib_query?bibcode=1997SPIE.3026...95N&db_key=PHY. [réf. du : 06 jan 2006]. Format PDF. 167, 219

[Nog97] D. NOGUET : A massively parallel implementation of the watershed based oncellular automata. pages 42–52, 1997. Disponible sur : http://ieeexplore.ieee.org/iel3/4817/13318/00606811.pdf?tp=&arnumber=606811&isnumber=13318. [réf. du : 12 jul 2005]. Format PPT. 28

[Nog98] Dominique NOGUET : *Architectures parallèles pour la morphologie mathématique géodésique*. Thèse de doctorat, Institut National Polytéchnique de Grenoble, janvier 1998. Disponible sur : http://tel.ccsd.cnrs.fr/documents/archives0/00/00/30/40/tel-00003040-00/tel-00003040.pdf. [réf. du : 12 jul 2005]. Format PDF. ISBN (paperback) : 2-913329-23-3 ; ISBN (electronic format) : 2-913329-23-3 ; Thèse de Doctorat INPG, Spécialité Microélectronique. 28, 194

[Nog02] Aleksey NOGIN : A Review of Theorem Provers. février 2002. Disponible sur : http://www.cs.cornell.edu/Nuprl/PRLSeminar/PRLSeminar01_02/Nogin/PRLseminar7b.pdf. [réf. du : 7 may 2006]. Format PDF. février 2002. 195

[Nor01] Per NORDLOW : Implementation Aspects Of Image Processing. Mémoire de D.E.A., Institutionen for systemteknik, Department of Electrical Engineering, Linkopings universitet, Sweden, mars 2001. Disponible sur : http://www.cvl.isy.liu.se/ScOut/Masters/Papers/Ex3088.pdf. [réf. du : 16 feb 2006]. Format PDF. 43, 50

[NSG05a] Samuel NAFFZIGER, Blaine STACKHOUSE et Tom GRUTKOWSKI : The Implementation of a 2-core, Multi-threaded Itanium Family Processor. *ISSCC 2005, San Francisco*, mars 2005. Disponible sur : http://www.ewh.ieee.org/r5/denver/sscs/Presentations/2005.03.Naffziger.pdf. [réf. du : 2 fev 2006]. Format PDF. 19, 20

[NSG05b] Samuel NAFFZIGER, Blaine STACKHOUSE et Tom GRUTKOWSKI : The Implementation of a 2-core, Multi-threaded Itanium Family Processor. Présentation. ISSCC 2005, San Francisco, mars 2005. Disponible sur : http://www.ewh.ieee.org/r5/denver/sscs/Presentations/2005.03.Montecito1.pdf. [réf. du : 2 fev 2006]. Format PDF. 19, 43

[NVi05] NVIDIA : Fast Texture Downloads and Readbacks using Pixel Buffer Objects in OpenGL, Technical Brief. août 2005. Disponible sur : http://download.nvidia.com/developer/Papers/2005/Fast_Texture_Transfers/Fast_Texture_Transfers.pdf. [réf. du : 14 may 2006]. Format PDF. août 2005. 185, 186, 209

[NVi06] NVIDIA : NVidia GeForce 7 Series GPUs Specifications. 2006. Disponible sur : http://www.nvidia.com/object/7_series_techspecs.html. [réf. du : 9 mar 2006]. Format HTML. 2006. 22

[OLG+05] John D. OWENS, David LUEBKE, Naga GOVINDARAJU, Mark HARRIS, Jens KRÜGER, Aaron E. LEFOHN et Timothy J. PURCELL : A Survey of General-Purpose Computation on Graphics Hardware. *In Eurographics 2005, State of the Art Reports*, pages 21–51, août 2005. Disponible sur : http://graphics.idav.ucdavis.edu/publications/func/return_pdf?pub_id=844. [réf. du : 12 jul 2005]. Format PDF. 22, 44, 56

[Ope06] OPENMP : OpenMP Web site. 2006. Disponible sur : http://www.openmp.org. [réf. du : 21 may 2006]. Format HTML. 2006. 50

[ORK+02] John D. OWENS, Scott RIXNER, Ujval J. KAPASI, Peter MATTSON, Brian TOWLES, Ben SEREBRIN et William J. DALLY : Media Processing Applications on the Imagine Stream Processor. *Proceedings of the 2002 International Conference on Computer Design*, 2002. Disponible sur : http://www.iccd-conference.org/proceedings/2002/17000295.pdf. 37, 46

[Owe04] John OWENS : GPUs : Engines for Future High-Performance Computing. 2004. Disponible sur : http://www.ece.ucdavis.edu/~jowens/talks/owens-hpec04-gpgpu.pdf. [réf. du : 12 jul 2005]. Format PDF. 2004. 22, 23

[Pel93] Susanna PELAGATTI : *A methodology for the development and the support of massively parallel programs*. Thèse de doctorat, Dipartimento di Informatica, Universita' di Pisa, mars 1993. Disponible sur : ftp://ftp.di.unipi.it/pub/Papers/susanna/thesis.ps.gz. [réf. du : 12 oct 2005]. Format PS.GZ. 26

[PJ00] Stelian PERSA et Pieter JONKER : Real Time Image Processing Architecture for Robot Vision. 2000. Disponible sur : http://www.ph.tn.tudelft.nl/People/pieter/pdfs/stelian_PE_SPIE2000.pdf. [réf. du : 12 jul 2005]. Format PDF. 2000. 65

[Pri06] Marc PRIEUR : AMD Torrenza, co-proc pour Opteron. juin 2006. Disponible sur : http://www.hardware.fr/news/imprimer/8194/. [réf. du : 06 jun 2006]. Format HTML. juin 2006. 194

[Rey06] John REYNOLDS : NVIDIA's GeForce 7800 GTX, preview. 2006. Disponible sur : http://www.simhq.com/_technology/technology_056a.html. [réf. du : 06 feb 2006]. Format HTML. 2006. 43

[RM00] Jos B. T. M. ROERDINK et Arnold MEIJSTER : The Watershed Transform : Definitions, Algorithms and Parallelization Strategies. *Fundamenta Informaticae*, 41:178–228, 2000. 28

[RS01] Martin RUMPF et Robert STRZODKA : Using Graphics Cards for Quantized FEM Computations. *Proceedings VIIP01*, pages 193–202, 2001. 28

[Rys05] Ondrej RYSAVY : *Specifying and reasoning in the calculus of objects*. Thèse de doctorat, Faculty of Information Technology, Brno University of Technology, 2005. Disponible sur : http://www.fit.vutbr.cz/research/view_pub.php?id=7921. [réf. du : 7 may 2006]. Format HTML. 195

[Sas02] Raphaël SASPORTAS : *Etude d'architectures spécifiques aux applications d'analyse d'image par morphologie mathématique*. Thèse de doctorat, Ecole Nationale Supérieure des Mines de Paris, octobre 2002. 28, 46

[SBJ96] Pierre SOILLE, Edmond J. BREEN et Ronald JONES : Recursive Implementation of Erosions and Dilations along discrete lines at arbitrary angles. 1996. Disponible sur : http://citeseer.ist.psu.edu/96171.html. [réf. du : 06 jan 2005]. Format PDF, PS. 167

[SDR04] R. STRZODKA, M. DROSKE et M. RUMPF : Image Registration by a Regularized Gradient Flow - A Streaming Implementation in DX9 Graphics Hardware. *Computing*, page 18, 2004. Disponible sur : http://numerik.math.uni-duisburg.de/research/papers/public/DrRuSt04.pdf. Format PDF. 28

[Sei04] Chris SEITZ : Evolution of GPUs. Présentation. Perfect Kitchen Art, 2004. 23

[Ser88] Jean SERRA : *Image Analysis and Mathematical Morphology, Volume 2 : Theoretical Advances*, volume 2. Academic Press, 3rd édition, 1988. Centre de Morphologie Mathématique, Ecole Nationale Supérieure des Mines de Paris, France. 84

[Ser89] Jean SERRA : *Image Analysis and Mathematical Morphology, Volume 1*, volume 1. Academic Press, 3rd édition, 1989. Centre de Morphologie Mathématique, Ecole Nationale Supérieure des Mines de Paris, France. 84, 86

[Ser00] Jean SERRA : Cours de la morphologie mathématique - Chapitre 6 : Connexion et fonctions numériques. Présentation. Cours de la morphologie mathématique, 2000. 115

[Sho51] William SHOCKELEY : Circuit Element Utilizing Semiconductive Material. septembre 1951. Disponible sur : http://www.bellsystemmemorial.com/pdf/02569347.pdf. [réf. du : 22 mar 2006]. Format PDF. septembre 1951. 23

[Ski92] D.B. SKILLICORN : The Bird-Meertens Formalism as a Parallel Model. Rapport technique, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1992. Disponible sur : http://ftp.qucis.queensu.ca/TechReports/Reports/1992-332.pdf. 27

[Ski98] David B. SKILLICORN : A Taxonomy for Computer Architectures. *IEEE COMPUTER*, 21(11):46–57, novembre 1998. Disponible sur : http://dx.doi.org/10.1109/2.86786. [réf. du : 12 jul 2005]. Format PDF. Dept. of Comput. & Inf. Sci., Queen's Univ., Kingston, Ont., Canada ; ISSN : 0018-9162. 27, 31

[Soi03] Pierre SOILLE : *Morphological Image Analysis : Principles and Applications*. Springer-Verlag, second edition édition, 2003. Disponible sur : http://portal.acm.org/citation.cfm?id=773286. [réf. du : 07 mar 2006]. Format citation HTML. 84, 85, 147, 167

[Spi03] John SPITZER : Graphics Performance Optimisation. Présentation. GDCE 2003, 2003. Disponible sur : http://developer.nvidia.com/docs/IO/8343/Performance-Optimisation.pdf. [réf. du : 13 may 2006]. Format PDF. 183

[SRU01] Jurij SILC, Borut ROBIC et Theo UNGERER : Asynchrony in parallel computing : from dataflow to multithreading. *Progress in Computer Research*, 1:1–33, 2001. Disponible sur : http://www.informatik.uni-augsburg.de/~ungerer/JPDCPdataflow.pdf. [réf. du : 29 mar 2006]. 47

[ST04] Robert STRZODKA et Alexandru TELEA : Generalized Distance Transforms and Skeletons in Graphics Hardware. pages 221–230, 2004. Disponible sur : http://numerik.math.uni-duisburg.de/research/papers/public/StTe04skeletons.pdf. 28

[Sta06] William STALLINGS : *Computer Organization and Architecture Designing for Performance*. Pearson Prentice Hall, seventh édition, 2006. 47, 48, 207

[Str02] R. STRZODKA : Virtual 16 Bit Precise Operations on RGBA8 Textures. *VMV*, page 9, novembre 2002. 28

[Str04] Robert STRZODKA : *Hardware Efficient PDE Solvers in Quantized Image Processing*. Thèse de doctorat, Universitaet Duisburg-Essen, 2004. Disponible sur : http://www.ub.uni-duisburg.de/ETD-db/theses/available/duett-02242005-000216/unrestricted/Strzodkadiss2004.pdf. [réf. du : 19 jul 2005]. Format PDF. 23, 28, 38

[Sup02] SUPERH : SH-5 product brief, 2002. Disponible sur : http://www.superh.com. [réf. du : 15 dec 2002]. Format PDF. document code : 05-PB-10001 V1.0. 43

[TBG+02] Xinmin TIAN, Aart BIK, Milind GIRKAR, Paul GREY, Hideki SAITO et Ernesto SU : Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology : Implementation and Performance. *In Intel Technology Journal (2002, Volume 06 Issue 01)*[Int02], pages 36–46. Disponible sur : ftp://download.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf. [réf. du : 31 mar 2006]. Format PDF. 50

[TC05] Pedro TRANCOSO et Maria CHARALAMBOUS : Exploring Graphics Processor Performance for General Purpose Applications. 2005. Disponible sur : http://216.228.112.225:8080/t/340/703432/225/0/. [réf. du : 12 jul 2005]. Format PDF. 2005. 56

[To95] Hing Wing TO : *Optimising the Parallel Behaviour of Combinations of Program Components*. Thèse de doctorat, University of London, Imperial College of Sciences, Technology and Medicine, Department of Computing, septembre 1995. Disponible sur : http://hpc.doc.ic.ac.uk/environments/coordination/papers/hwt.ps.gz. [réf. du : 18 oct 2005]. Format PS.GZ. 26

[TP05] Damien TRIOLET et Marc PRIEUR : NVIDIA GeForce 7800 GTX. juin 2005. Disponible sur : http://www.hardware.fr/art/lire/574/. [réf. du : 9 mar 2006]. Format HTML. juin 2005. 22, 43

[Tra05a] TRANSMETA : Crusoe : Architecture, 4 Instruction Issue, 128-bit VLIW Engine. 2005. Disponible sur : http://www.transmeta.com/crusoe/vliw.html. [réf. du : 15 mar 2006]. Format HTML. 2005. 37, 42

[Tra05b] TRANSMETA : Efficeon : Architecture, High Performance 8 Instruction Issue, 256-Bit VLIW Engine. 2005. Disponible sur : http://www.transmeta.com/efficeon/vliw.html. [réf. du : 15 mar 2006]. Format HTML. 2005. 37, 42

[Tra05c] TRANSMETA : Efficeon Model TM8800 Processor - Specifications. 2005. Disponible sur : http://www.transmeta.com/efficeon/efficeon_tm8800.html. [réf. du : 20 mar 2006]. Format HTML. 2005. 43, 44

[Tur37] A. M. TURING : Computability and λ-Definability. *Journal of Symbolic Logic*, 2(4):153–163, décembre 1937. 59

[Unk06] Author UNKNOWN : Eleven Reasons to use Haskell as a Mathematician. janvier 2006. Disponible sur : http://sigfpe.blogspot.com/2006/01/eleven-reasons-to-use-haskell-as.html. [réf. du : 17 feb 2006]. Format HTML. janvier 2006. 60

[Ven03] Suresh VENKATASUBRAMANIAN : The Graphics Card as a Stream Computer. *SIGMOD-DIMACS Workshop on Management and Processing of Data Streams*, 2003. Disponible sur : http://www.research.att.com/~suresh/papers/mpds/mpds.pdf. [réf. du : 31 jul 2005]. Format PDF. 56

[vH92] Marcel van HERK : A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels. *Pattern Recogn. Lett.*, 13(7):517–521, 1992. Disponible sur : http://dx.doi.org/10.1016/0167-8655(92)90069-C. [réf. du : 28 mar 2006]. Format PDF. 167, 168

[Wik05a] WIKIPEDIA : Instructions per second. octobre 2005. Disponible sur : http://en.wikipedia.org/wiki/Million_instructions_per_second. [réf. du : 14 nov 2005]. Format HTML. octobre 2005. 43

[Wik05b] WIKIPEDIA : Lambda calcul. 2005. Disponible sur : http://fr.wikipedia.org/wiki/Lambda-calcul. [réf. du : 14 oct 2005]. Format HTML. 2005. 59

[Wik05c] WIKIPEDIA : Pseudocode. 2005. Disponible sur : http://en.wikipedia.org/wiki/Pseudocode. [réf. du : 14 oct 2005]. Format HTML. 2005. 26

[Wik05d] WIKIPEDIA : Thèse de Church-Turing. 2005. Disponible sur : http://fr.wikipedia.org/wiki/Thèse_de_Church-Turing. [réf. du : 14 oct 2005]. Format HTML. 2005. 59

[Wik06a] WIKIPEDIA : Abou-Jafar-Muhammad-Ibn-Musa-al-Khuwarizmi. 2006. Disponible sur : http://fr.wikipedia.org/wiki/Al-Khwarizmi. [réf. du : 30 apr 2006]. Format HTML. 2006. 177

[Wik06b] WIKIPEDIA : Algèbre. 2006. Disponible sur : http://fr.wikipedia.org/wiki/Algèbre. [réf. du : 10 may 2006]. Format HTML. 2006. 177

[Wik06c] WIKIPEDIA : Curryfication. 2006. Disponible sur : http://fr.wikipedia.org/wiki/Curryfication. 2006. 61

[Wik06d] WIKIPEDIA : Dhrystone. 2006. Disponible sur : http://en.wikipedia.org/wiki/Dhrystone. [réf. du : 04 apr 2006]. Format HTML. 2006. 205

[Wik06e] WIKIPEDIA : Donald Ervin Knuth. 2006. Disponible sur : http://fr.wikipedia.org/wiki/Donald_Knuth. [réf. du : 30 apr 2006]. Format HTML. 2006. 178

[Wik06f] WIKIPEDIA : FLOPS. novembre 2006. Disponible sur : http://en.wikipedia.org/wiki/FLOPS. [réf. du : 14 nov 2005]. Format HTML. novembre 2006. 205

[Wik06g] WIKIPEDIA : Haskell Brooks Curry. 2006. Disponible sur : http://en.wikipedia.org/wiki/Haskell_Curry. [réf. du : 9 apr 2006]. Format HTML. 2006. 60

[Wik06h] WIKIPEDIA : Multimédia. janvier 2006. Disponible sur : http://fr.wikipedia.org/wiki/Multimedia. [réf. du : 16 feb 2006]. Format HTML. janvier 2006. 29, 43, 206

[Wik06i] WIKIPEDIA : PlayStation 3. 2006. Disponible sur : http://en.wikipedia.org/wiki/PS3. [réf. du : 14 mar 2006]. Format HTML. 2006. 37, 43

[Wik06j] WIKIPEDIA : Processus léger. 2006. Disponible sur : http://fr.wikipedia.org/wiki/Thread. [réf. du : 30 mar 2006]. Format HTML. 2006. 47

[Wik06k] WIKIPEDIA : Wikipedia. 2006. Disponible sur : http://fr.wikipedia.org/wiki/Wikipédia. [réf. du : 22 apr 2006]. Format HTML. 2006. 29

[Wik06l] WIKIPEDIA : XBOX 360. 2006. Disponible sur : http://en.wikipedia.org/wiki/Xbox_360. [réf. du : 14 mars 2006]. Format HTML. 2006. 37, 43

[Wil94] Herbert S. WILF : *Algorithms and Complexity*. 1st édition, 1994. Disponible sur : http://www.math.upenn.edu/~wilf/AlgoComp.pdf. [réf. du : 22 apr 2006]. Format PDF. University of Pennsylvania, Philadelphia. 178, 179

[Wlo03] Matthias WLOKA : Batch, Batch, Batch : What Does It Really Mean ? Présentation. Game Developers Conference 2003, 2003. Disponible sur : http://developer.nvidia.com/docs/IO/8230/BatchBatchBatch.ppt. [réf. du : 13 may 2006]. Format PPT. 187

[WNDS99] Mason WOO, Jackie NEIDER, Tom DAVIS et Dave SHREINER : *OpenGL Programming Guide*. Addison-Wesley, third edition édition, octobre 1999. 51, 54, 55

[Yan97] Jin YANG : *Co-ordination Based Structured Parallel Programming*. Thèse de doctorat, University of London, Imperial College of Sciences, Technology and Medicine, Department of Computing, octobre 1997. Disponible sur : http://hpc.doc.ic.ac.uk/environments/coordination/papers/jy.ps. [réf. du : 18 oct 2005]. Format ps. 26, 73

[YW02] Ruigang YANG et Greg WELCH : Fast Image Segmentation and Smoothing Using Commodity Graphics Hardware. *Journal of Graphics Tools*, 2002. Disponible sur : http://www.cs.unc.edu/~stc/publications/Yang_jgt22003.pdf. [réf. du : 12 jul 2005]. Format PDF. 2002. 28

Cette page est blanche par intention

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z